EXHIBIT C

Flow Director CAM (FDC) High Level Design

Revision 1.33

May 10, 2002

Simon Knee

Revision History

Revision	Date	Author	Description of Changes
0.10	March 19, 2001	Simon Knee	Initial document.
0.20	March 19, 2001	Simon Knee	Preview release.
0.51	March 27, 2001	Simon Knee	General Updates / Fixes: • Changes made to 0.20 due to initial review by Pramod and Fazil. • Modified so that FDC can indicate when no processor cores / workspace IDs were available. • Modified so that the FDC indicates when an entry was created: this is required for the Dispatcher.
1.00	March 30, 2001	Simon Knee	General Updates / Fixes: • The TDFIDX and UPFIDX commands needed a response: so that the Dispatcher can see if it needs to issue a LUC update or tear down command. • Various updates after review by engineering team.
1.01	April 13, 2001	Simon Knee	General Updates / Fixes: • Increased the FDC Index from 6 to 7 bits in the FDC command responses. This allows for up to 16 processor cores with 16 workspaces each, or 32 processor cores with 8 workspaces. • Increased the workspace size from 2 to 4 bits in the FDC entry format and FDC command responses. This allows for up to sixteen workspaces per processor core. • Increased the Event Index field of the FDC command responses from 3 to 4 bits. This allows for up to sixteen event queue elements per processor core. • Increased the Processor core field to 5 bits, but this is limited to a maximum of 24 processor cores rather than 32. • Updated the register definitions section so that each bit field has a name. • Modified the GETEVENT command format and semantics so that a Processor core mask *can be* supplied. This will allow the Dispatcher to replicate events to multiple Processor cores (e.g. ARP). • Modified the FDC entry state diagram so that in the *PENDING* state we can receive UPFIDX and TDFIDX commands. The FDC response from these commands is to do nothing: the Dispatcher must back off and re-issue. This was required to ensure correct workspace valid bit setting for packet / interface events received in the *PENDING* state. • Increased the FDC size from 64 entries to 128. This allowed the associated known issue to be removed, as it has now been resolved.
1.02	April 30, 2001	Simon Knee	General Updates / Fixes: • Reduced the size of the LUC_FDC_Bus and Dispatcher_FDC_Bus to 8-bits. • Some sub sections of "Managing the Processor core Event Queues / Workspaces" were using an incorrect order for indexes: `ws_available` calculation and `event_bits` were incorrect. • *Pending* State Transitions table had an incorrect Flags value for the TDFIDX command. • Correct the section on FDC Bulk Data Transfer metric: we know require 1,233,000 command combinations per second, rather than 850,000. Added a section on FDC Performance Based on the Timer Metric. • Modified bulk throughput metric to allow for 10Gbits full duplex. • *Received* State Transitions indicated that a CRTIMER command is invalid. This is not true: the LUC may not have seen the request in

Revision	Date	Author	Description of Changes
			transit for this flow, so it may try to create a timer entry. The response simply indicates that this entry is in the *RECEIVED* state, and it is not an error. • Added a section to describe fatal and non-fatal responses from the FDC. Modifications due to a review by Robert Johnson: • The CRTIMER response included the event index, processor core and workspace ID. This was not correct, and those fields have been removed. • The UPFIDX and TDFIDX commands included the processor core in the request. This is not required as the commands supply the FDC index, and the FDC entry includes the processor core.
			• The Event Mask of the GETEVENT request format was reduced from 32 to 24 bits (only twenty-four processor cores max), and was moved into the second word. • *Received* State Transitions table incorrectly specified that the FDC Index, processor core, event index and workspace ID must be set for a UPFIDX or TDFIDX response. • Added a "Next State" assignment to each protocol transition table to make it more obvious what the next state is. • *Pending* State Transitions table had an incorrect Flags value for the SERVTIMER command. • Swapped the position of the FDC Index and Event Index in the FDC responses to make it more compatible with the Message Bus formats. • Added registers to count the number of FDC commands received, number of times we ran out of CAM space, number of times we could not allocate a processor core. This is to aid debug and tuning. Modifications due to a review by Perry Virjee: • Flags field was mislabeled in FDC entry format, `event_mask` had 32 instead of 64 entries in "Managing the Processor core Event Queues / Workspaces" section. • Tidied up section on "Flow Direct CAM Size" so that it more clearly defined the FDC size • RELEVENT, TDFIDX, UPFIDX request formats were using 3-bits for the Event Index instead of 4.
1.03	September 24, 2001	Simon Knee	Many modifications due to Perry Virjee and Robert Johnson, and various other bug fixes: • Update the FDC block diagram, updated bus interface definitions. • Modified the "Expected Performance" section to say that we do not expect all performance metrics to be met at the same time: they are disjoint. • Modified formats and text to allow for 116-bit flow key. • Modified the Flow Director State Diagram (Figure 5) and associated sections (2.6.3, 2.6.4, 2.6.6, 2.6.8) to show: o *CHECKED IN* state value is 0xxxx. o LFKCREATE in *CHECKED IN* with no space remains in *CHECKED IN*. o CRTIMER in *RECEIVED* transitions to *RECEIVED* state. o CRTIMER in *TIMER* transitions to *TIMER* state. o CRTIMER in *DELETE* transitions to *DELETE* state. • Modified the *TIMER* state transitions table (Table 13) so that the CRTIMER command no longer sets the invalid command bit.

Revision	Date	Author	Description of Changes
			• The description of the flags field of section 3.4.4 was not correct. • Added an Event Type field to the SERVTIMER command of section 3.4.7. • Added a MASK register (see section 3.5.5). • Added an implementation guide for the round robin algorithm (see section 2.4.10). • Updated the Flow Director CAM overview figure (Figure 1). • Added section 2.1.1 on *Processor Core Numbering*. Modified text throughout to reference this section. • Added debug read/write access of the CAM. See sections 3.5.9 and 3.5.10. • Modified sections 2.4 and 3.5.11 to show how the length of the event queue is determined.
			• Replaced with EQLENGTH register with WSLENGTH. We no longer need to be given the length of the event queue in another register – we can calculate it via other means. • Fixed the event size at 256 bytes, reduced the maximum number of events per Processor Core to 8. • Introduced Event Type sub-structure in section 2.4.2.1. This allows the number of event masks that the FDC must store to be reduced from 64 to 32. • Modified Figure 3, Flow Director CAM Entry Format so that the bit positions in a 32-bit word are easier to see. This is only cosmetic – no logical changes were made. • Added section 2.4.10 on workspace sizes, and section 2.4.11 on event queue lengths.
1.20	October 10, 2001	Simon Knee	Modifications made as described below. • Modifications through out to allow for interface core / protocol core integration. Changed the name of the *Other Event Queue* on the Dispatcher to *ND Event Queue*. • Gave a RELEVENT command a response format. See Figure 13. • Corrected the section on FDC size (2.3) so that the maximum number of processor cores is 15. Adjusted other bitmaps to reflect this. Added a section about B10 versus S10 FDC size. • Modified the processor core and workspace allocation algorithms to allow for protocol and interface core management. See section 2.4. This also required modification of the FDC command formats and responses. See sections 3.4.2 through 3.4.9. Also had to modify the state transition tables of sections 2.6.3 through 2.6.8. • Added section 2.1.4 to describe the difference between an event index and an event number. • In Table 11, a CRTIMER command in the DELETE state was causing an "error condition" of 5'b11111 to be returned. This should have been 5'b11110, i.e. not an error condition. • For RMFIDX, Table 10 and Table 11 incorrectly stated that an event number is to be released – only the workspace ID is released. • Made the CONTROL register read and write instead of write only (see section 3.5.1).
1.21	October 16, 2001	Simon Knee	• Modified Table 16: FDC Register Map so that it uses hexadecimal addresses that start at 0080. 0080 is the start of the FDC register block in the Dispatcher register map. • Added section 2.1.2 on Core Index values. Modified section 2.1.3 to use the term Core Index.

Revision	Date	Author	Description of Changes
			• Modified the FREEBUSY registers so that they are not indexed by Core ID, but instead are compressed together. See Table 16: FDC Register Map. • Modified the Processor Core / Event Index / Workspace ID selection algorithm of section 2.4.6 so that it more accurately reflects the real implementation algorithm. • Modified the variables of section 2.4.3 so that they are index by Core Index rather than Core ID. • Added text to section 3.4.3 to explain that a response to a GETEVENT does not include an Event Mode.
1.22	October 22, 2001	Simon Knee	Bug fixes: • Figure 8: GETEVENT Request Data Format was using a 24-bit Event Mask. Should be 15-bits. • Modified section 2.4.6 so that for stateless event processing we logically OR PC_EVENT_MASK and IC_EVENT_MASK. • Added text to the ALLOC_PC_EVENT bit of the EVENT_MASK register (section 3.5.12) saying that for Single Core Mode the software must set ALLOC_PC_EVENT to value 1 for all Event Types. Modified section 2.4.5 on Protocol Core / Event Index / Workspace ID allocation to take advantage of this. • Section 2.4.12 on *Resource Allocation Implementation Guide*: stated that in the RTL the *Round Robin Vector* for the Interface Core is always adjusted. • Section 2.4.2.1 on Event Type Substructure: said that if the MSB of the Event Type is set, and if the lower 5-bits are not a valid Core ID, then a Event Mask of all zeros will be used.
1.23	November 4, 2001	Simon Knee	Bug fixes: • Made it clear in that the PC_EVENT_MAKS and IC_EVENT_MASK must not overlap. See sections 2.4.4 and 3.5.12. • Modified the LFKCREATE and SERVTIMER responses (Figure 11Figure 17) to prevent fields crossing 32-bit boundaries. • Added the CAM_INIT bit to the CONTROL register (see section 3.5.4). • Figure 8: GETEVENT Request Data Format only showed 14-bits for the mask, should have been 15. • In Table 11: *Delete* State Transitions, the responses for CRTIMER and LFKCREATE were not correct. • In Table 13: *Timer* State Transitions, the response for CRTIMER was not correct. • Added section 3.6 on how to initialise the FDC. • Expanded section 2.3 on the FDC CAM size, and made the math more robust. • Section 2.4.10 on *Workspace Sizes* incorrectly said that the minimum workspace size is 256 bytes. It is 128 bytes. • Added text to sections 2.4.4 through 2.4.7 to make it clear that we must be able to detect a full FDC independent of whether workspaces / event indexes are available. Also modified those sections to make it clear when a full FDC should stop the allocation, and when it should not.
1.24	November 9, 2001	Simon Knee	Bug fixes: • PR 75 Fix: Made it clear in sections 3.5.9 and 3.5.10 that the FDC CAM access via the MMC is read-only. • PR 79 Fix: Made the INIT_EBITS values accessible via spare bits in the FREEBUSY register (see section 3.5.11).

Revision	Date	Author	Description of Changes
			• Removed the *FDC Request Queue* and *FDC Resp. Queue* from Figure 1: Flow Director CAM Overview as they do not exist and are not required.
1.25	December 3, 2001	Simon Knee	Bug fixes: • PR 124 Fix: Added a note in section 3.5.1 that the INV_MCC_ADDR bit of the *Dispatcher* STATUS register captures invalid MMC accesses to the FDC. • PR 120 Fix: CRTIMER command in the *DELETE* state is not valid. Figure 5: Flow Director State Diagram and Table 11: *Delete* State Transitions modified. • Added default values to all registers of section 3.5. • PR 139 Fix: *CHECKED IN* state value should be 5'b00000. Figure 5: Flow Director State Diagram and sections 2.6.3, 3.4.6 modified accordingly. • Table 8: *Check In* State Transitions, clarified that the *Event Count* is set to value one when a FDC entry is created. • PR 158 Fix: Increased the *Event Count* of Figure 3: Flow Director CAM Entry Format to 7-bits, added EV_OVFLOW to STATUS register, added checks to LFKCREATE processing in the *RECEIVED* and *PENDING* states. No need to check in the *TIMER* state since the *Event Count* must be zero in that case. • Added section 3.5.2 that explains that the FDC must process a register write in seven clock cycles or less. • Sections 3.1.1 and 3.2.1 on the Dispatcher FDC Bus and FDC Dispatcher Bus performance did not allow for the LUC commands/responses traversing them. • PR 161 Fix: Renamed the ERR bit of the STATUS register to UNREC_CMD_ERR and made it clear that this is only set when an invalid *FDC Command* value is received. Modified all state transition tables to show that for invalid commands in those states there is no action required, i.e. we do not set any status bits.
1.26	December 16, 2001	Simon Knee	Bug fixes: • PR 168 Fix: FDC now looks at the SMC_DISP_Almost_Full signal and modifies the response of a CRTIMER command in the *CHECKED IN* state. See section 2.7.
1.27	December 21, 2001	Simon Knee	Bug fix: • PR 255 Fix: Bit 2 of the STATUS register (DIS_SMC_DISP_BP) moved to bit 2 of the CONTROL register.
1.28	January 30, 2002	Simon Knee	Bug fixes: • PR 453 Fix: Changed the ADDR field of the CAM_ADDR register (section 3.5.9) from 8 to 7 bits. • Added register offsets to the section headings of sections 3.5.3 through 3.5.12.
1.29	February 19, 2002	Simon Knee	Bug fixes: • PR 643 Fix: Some register names changed to match Hardware Reference Manual.
1.30	March 26, 2002	Simon Knee	Bug fixes: • The header of the v1.29 FDC HLD said v1.28. Revised to version 1.30 to avoid confusion. No other changes were made.
1.31	April 18, 2002	Simon Knee	Bug fixes: • PR 1148 Fix: Registers are addresses 008D-008F should have been marked as N/A not Read / Write.

Revision	Date	Author	Description of Changes
1.32	April 29, 2002	Simon Knee	Bug fixes: • PR 1213 Fix: *Table 16: FDC Register Map* modified to show that CAM_DATA registers are read-only. • PR 1214 Fix: *Table 25: Free/Busy Register Bit Definitions* had an 8-bit default value for WBITS, should have been 16-bits.
1.33	May 10, 2002	Simon Knee	Bug fixes: • PR 1312 Fix: Modified *Table 19: Mask Register Bit Definitions* to show which bits were spare.

Table of Contents

List of Figures

List of Tables

1 Introduction

In this document we describe the high level operations of the Flow Director CAM. This document should describe all the necessary behaviour of the Flow Director CAM, without enforcing any particular implementation method.

It is likely that a single pass of reading this document is not sufficient to gather all information. This document should also be read in conjunction with other HLD documents that are tightly integrated with the operation of the FDC, e.g. Dispatcher, LUC, Processor Cores.

1.1 Related Documents

Document	Revision	Author
Astute Content Processor Architecture Presentation	N/A	Fazil Osman
Deadlock Analysis and Avoidance	1.20	Brian Petry
Dispatcher High Level Design	1.30	Simon Knee
Event Specification	1.22	Simon Knee
Host Message API (Level 3) High Level Design	1.23	Billy Oostra
Input Processing Unit (IPU) High Level Design	1.37	Bob Sefton
LookUp Controller (LUC) High Level Design	1.29	Simon Knee
ManageMent Controller High Level Design	1.11	Nitesh Mehta
Message Bus High Level Design	1.30	Mark Feinstein
Output Processing Unit (OPU) High Level Design	1.24	Bob Sefton
Packet Processor High Level Design	1.20	Octera
Protocol Cluster High Level Design	1.16	Kirk Larson
Queuing Model Trace of a Simple HTTP Request	1.00	Simon Knee
Socket Memory Controller High Level Design	1.12	Simon Knee
Some Statistical Plots of Web Traffic	1.00	Brian Petry
Scratchpad High Level Design	1.50	Charles Kaseff
SPI-4 Module High Level Design	1.02	Bob Sefton
TCP Processing Paths for the Content Processor	1.00	Simon Knee, Brian Petry
TCP/IP Algorithm Review	0.91	Brian Petry
TCP/IP Core Software: Functional Specification and Conformance Statement	1.10	Brian Petry
TCP/IP Core Software: High Level Design	WIP	Brian Petry

1.2 Overview

The Flow Director CAM (FDC) is used to ensure coherency between all of the processor cores. It ensures that if a processor core is processing a particular frame of a protocol's flow, then any frames that are received during that time are sent to the same core. This simplifies the task of maintaining coherency, and removes the need for any special semaphores or locking on the flow state.

The FDC manages the assignment and release of the processor cores, and runs in one of two modes:

- **Single Core Mode.** In this mode the FDC allocates a single processor core for each CAM entry.
- **Dual Core Mode.** In this mode the FDC allocates two processor cores for each CAM entry: a protocol core and an interface core. The protocol core and interface core must be on the same cluster.

A single configuration bit in the CONTROL register determines the mode of the FDC, i.e. we operate in either one mode or the other, and never switch between the two. The single core mode of operation allows all available processor cores to be used for processing events. The dual core mode allows the processor cores to be split into groups of protocol cores and interface cores.

This method of maintaining coherency means that for back-to-back frames from the same flow they always go to the same Processor Core or Cores, i.e. the maximum bandwidth of a single flow is limited by the processing speed of a single Processor Core. This necessitates the use of at least as many flows as there are processor cores to get maximum throughput[1].

The FDC also manages the assignment and release of the processor cores event queue elements and workspace IDs. In the case of the dual mode of operation, this involves the allocation of two processor cores, an event queue element and two workspace IDs.

As illustrated by Figure 1, the FDC has one interface with the Dispatcher. However, in this document we refer to either the Dispatcher or the LUC being the originator of FDC commands. The reason for this is that although the FDC only has one interface, the Dispatcher takes FDC commands from the LUC on the LUC_FDC bus, communicates them to the FDC, and sends the response back on the FDC_LUC bus.

The ACP uses the ManageMent and Control (MMC) interface for microprocessor access to registers and memory on each block. In Figure 1 we use a *Slave MMC Mux* to split a single MMC interface between the Dispatcher and the FDC. Note that this *Slave MMC Mux* is part of the Dispatcher, and as such is not described in the FDC HLD. The consequence of this is that the FDC is not assigned a direct MMC ID. Instead all FDC registers are accessed via the address block of the Dispatcher. See the Dispatcher HLD and MMC HLD for further details.

[1] It is likely that we will require more flows than the number of processor cores to get maximum throughput: various pipelines need to be kept busy, e.g. the LUC. Perhaps a more reasonable number is twice the number of processor cores.

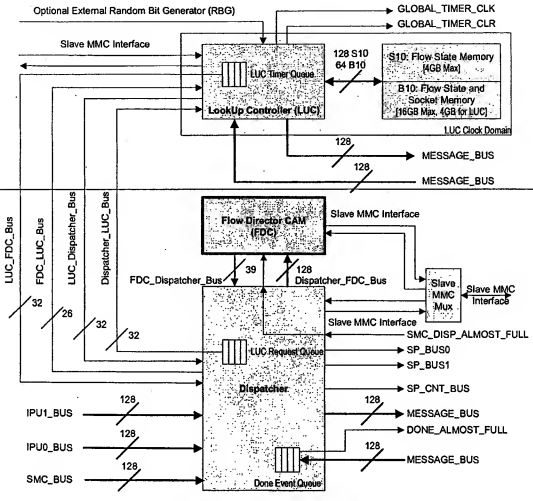

Figure 1: Flow Director CAM Overview

The Flow Key indexes each entry in the FDC. For TCP termination this flow key will be a 116 bit value consisting of the {IP Destination Address, IP Source Address, Destination Port, Source Port, IP Protocol, Receive Interface} tuple. Each entry in the FDC also has an associated payload that contains the assigned protocol core, the protocol core workspace ID, the assigned interface core, the interface core workspace ID, an event count and other flags. These payload components will be described in full detail later in the document.

In the simplest form, the operation of the FDC is easy to describe. For each packet or application event that the Dispatcher processes, it uses the FDC to determine if the event's flow key is present. If the event's flow key is present then the FDC payload describes which processor core pair are currently processing this flow. The event is then passed to one of those cores, and the event count is incremented. If the event's flow key is not found in the FDC then the Dispatcher must create an entry, assign it one or two processor cores, and assign it an event count of one. We can therefore see that the FDC ensures that while any event in a flow is being processed, all events go to the same processor core pair.

The event count of the FDC is used to determine when a FDC entry can be removed. Each time a processor core pair has finished processing an event it sends a "Done" message to the Dispatcher. The Dispatcher then finds the event count of the associated FDC entry, and decrements it. If this event count reaches zero

then there are no outstanding events in transit, so the Dispatcher signals to the LUC that the flow state can be written back. After the LUC has written back the flow state it removes the entry from the FDC. It is important to note that even though a pair of processor cores may be allocated, only one of those pairs replies with a stateful done event[2].

In order to process an event a processor core needs two pieces of information: the event itself, and the workspace ID associated with that event. Each processor core has an event queue that is used to receive events from the Dispatcher. In order to maintain free/busy information, the FDC assigns each position in this queue an event number. Each processor core also has some number of workspace IDs, which the FDC also maintains free/busy information.

Timer events also use the FDC. When the Timer Control Unit (TCU) determines that a timer has expired, it creates an entry in the FDC with the associated flow key[3]. The reason a timer entry is created is to force a synchronisation point between the LUC and the Dispatcher: packet and interface events must not be processed before the timer expired event. Note however that it does not assign a processor core pair. Instead the Dispatcher is given a "Timer Expired" message, and it updates the FDC entry with the assigned processor core pair. This timer event is now treated in a similar fashion to the packet and application event handling described above.

1.3 End Cases and Race Conditions

Given the description in section 1.2 it would seem that the FDC is a relatively simple block to describe. However, there are a number of end cases and race conditions that we must take care of:

1. What happens if a frame is received after the Dispatcher has issued an update request to the LUC. We must somehow inform the LUC that a frame has arrived and that it should not delete the entry from the FDC.

2. There is a small period of time where a timer has expired on a flow, but the Dispatcher has not yet read the "Timer Expired" message. What does the Dispatcher do if it receives a frame during this period of time, and how will the FDC signal that this condition has occurred.

To overcome these issues, and others, the FDC assigns a state to each entry, and uses a state machine to determine the correct action to perform. This state machine is described in more detail in section 2.6.

2 Functional Operation

2.1 Numbering Schemes

In the following sections we describe the numbering schemes used for the Core ID, Core Index and Core Bitmaps. Note that the schemes for Core ID, Core Index and Core Bitmaps are identical to those for the Dispatcher.

2.1.1 Core ID

We use the term Core ID to refer to a number that describes a processor core, i.e. it describes either a protocol core or an interface core in the *system*.

Cores are allocated to a cluster. There are three clusters in total, with each cluster containing five cores. Figure 2 illustrates the format that is used to create a Core ID. The Core ID is basically constructed from two

[2] The other Processor Core may reply with a stateless done event. It would do this if it wanted to free up its Event Index.
[3] The Timer Control Unit (TCU) is part of the LookUp Controller (LUC), and as such they share the same bus to the FDC. In fact, the TCU and LUC are so closely integrated that they can be considered the same device.

bits of Cluster Number, followed three bits of Core Number. This Core ID format is the format used on the Message Bus. Note that for three clusters with five cores per cluster, the valid Core ID values are 0, 1, 2, 3, 4, 8, 9, 10, 11, 12, 16, 17, 18, 19, 20. Specifically, the Core ID values 5, 6, 7, 13, 14 and 15 are not valid.

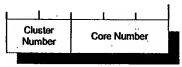

Figure 2: Core ID Format

2.1.2 Core Index

A Core Index is an encoding of the Core ID, as shown in Table 1. A Core Index provides a number in the range 0 through 14 with no holes, unlike a Core ID that has holes at 5, 6, 7, 13, 14, and 15.

Core Index	Core ID
0	0
1	1
2	2
3	3
4	4
5	8
6	9
7	10
8	11
9	12
10	16
11	17
12	18
13	19
14	20

Table 1: Core Index to Core ID Mapping

2.1.3 Core Bitmaps

Through out the FDC we often need to keep a bitmap of cores. The question is in this bitmap what Core ID does bit i represent? One obvious choice is that bit i represents Core ID i. The problem with this is that not all Core ID's are valid, so the bitmap would be larger than it really needs to be. Since bitmap width is important, the FDC uses another encoding in its Core Bitmaps. To solve this the FDC uses a bitmap such that bit i represents Core Index i. Table 1 can then be used to convert this Core Index into a Core ID.

2.1.4 Event Index vs. Event Number

We use the term Event Index to refer to a position in a Processor Cores Event Queue. As far as the FDC is concerned, Events are a fixed maximum size of 256 bytes, and there are a maximum of eight such events in an Event Queue.

From the viewpoint of the Protocol Cluster, it indexes its Event Queue memory using a 128 byte chunks using an Entry Index. The FDC translates Event Numbers into Event Indexes simply by shifting the Event Number left by one.

The only part of the ACP that understands the concept of an Event Number is the FDC. All other parts of the ACP communicate using Event Indexes.

2.2 Flow Director CAM Entry Format

Figure 3 illustrates the format of an entry in the FDC CAM. Table 2 describes the various fields. Note that duplicate entries are not allowed in the CAM, i.e. a given Flow Key either has a single hit in the CAM or is not found. In fact, due to the commands defined in section 2.4.10, it is not possible to create two entries with the same Flow Key. This feature of only one CAM entry per Flow Key is essential to the operation of the FDC.

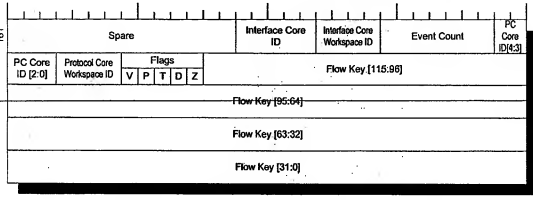

Figure 3: Flow Director CAM Entry Format

Field		Description
Flow Key		This is the key that is being used in the lookup. For TCP termination this is the tuple of the IP DA, IP SA, Destination Port, Source Port, IP Protocol and Receive Interface.
Interface Core ID		This indicates the Interface Core ID that has been assigned for processing this flows outstanding interface events. See section 2.1.1 for information on Core IDs. For single core mode operation this field is not valid.
Interface Core Workspace ID		This indicates the workspace ID within the above interface core that has been assigned to this flow. For single core mode operation this field is not valid.
Event Count		This is the count of outstanding events for this entry.
PC Core ID		This indicates the Protocol Core ID that has been assigned for processing this flows outstanding protocol events. See section 2.1.1 for information on Core IDs.
Protocol Core Workspace ID		This indicates the workspace ID within the above protocol core that has been assigned to this flow.
Flags	Valid	1 if the entry is valid (full), and 0 if the entry is invalid (empty).
	Pending	1 if the entry received a frame or interface event after an update was issued, zero otherwise.
	Timer	1 if the entry was created due to a timer expiration, zero otherwise.
	Deleting	1 if the entry has had a "LUC Teardown" issued against it.
	Zero Count	1 if the entry has an Event Count of zero.

Table 2: Flow Director CAM Fields

2.2.1 Flow Director Event Count Size

The *Event Count* size of the CAM entry is 7-bits wide. This allows for a maximum of 127 outstanding events for a single flow. The maximum number of outstanding events for a single flow is calculated as follows:

1. The Protocol Core can store eight events on its Dispatcher input queue.
2. The Interface Core can store eight events on its Dispatcher input queue.
3. The Protocol Core can store sixteen events on its Inter-Core messages input slots. This is done by moving a Dispatcher event into the Inter-Core bucket, and then replying with a stateless done to release the input event queue.

4. The Interface Core can store sixteen events on its Inter-Core messages input slots.
5. Software queuing inside the Protocol or Interface core accounts for another N_{SOFTQ} events. This amount is completely defined by software. In fact, it is expected that $N_{SOFTQ} = 0$ in most cases.

Summing these up gives the total number of outstanding events as $8 + 8 + 16 + 16 + N_{SOFTQ} = 48 + N_{SOFTQ}$. The problem is how do we limit N_{SOFTQ} such that $48 + N_{SOFTQ}$ never exceeds 127? The answer is that we cannot since there is no way to prevent software from queuing events internally. Instead we will detect when the *Event Count* field overflows, setting a bit in the FDC status register when it does[4]. If this bit is set then the FDC is no longer in a consistent state, and it must be reset.

2.3 Flow Director CAM Size

The size of the FDC is determined by how many flows we can process at once, since each flow that is processed requires an FDC entry[5]. This is determined by two factors:

1. Each flow requires a workspace. The number of flows being processed is therefore less than the total number of workspaces available. In the case of dual core mode, each flow requires both a Protocol Core workspace and an Interface Core workspace, therefore the number of flows being processed is less than the minimum of the Protocol or Interface Core workspaces.
2. Each workspace is pointed to by either an Event from the Dispatcher, or an Inter-Core message. Therefore, you cannot process more flows than there are the total number of Event Indexes and Inter-Core message slots available. For an example of how an Inter-Core message can be used to reference a workspace, the reader is directed to the *Dual Mode Stateful Event, Early PC Event Release* example of the *Dispatcher HLD*.

2.3.1 Analysis

In this section we provide a more robust analysis of the size requirements for the FDC. Table 3 introduces the variables used.

Variable	Description
C	The total number of Processor Cores.
C_p	The number of Protocol Cores.
C_i	The number of Interface Cores.
W	The number of workspaces that are available.
W_p	The number of workspace IDs on a single Protocol Core.
W_i	The number of workspace IDs on a single Interface Core.
E	The number of Event Indexes on a single Processor Core.
F	The number of entries in the FDC.

Table 3: FDC Size Variables

2.3.1.1 Single Core Mode

In single core mode we allocate a single workspace to a single Processor Core. That Processor Core processes the workspace and then checks it in. The size of the FDC need therefore be no larger than the total number of workspaces or the total number of events. Note that we add eight to this number to allow for the LUC creating timer entries. Entries that are in the timer state do not require a workspace ID or Event Index. We keep eight entries available for the LUC since that is the maximum number of entries it can expire in a single LUC tick.

[4] Note that the FDC responds as if the *Event Count* had not wrapped, i.e. it completely ignores the condition. It is up to a Processor Core to see the bit in the status register and act accordingly.
[5] Note that in dual core mode, even though we assign a protocol core and an interface core, only one FDC entry is required.

$$F' = \min(W \times C, E \times C) + 8 \quad [1]$$

Due to the hardware implementation, the number of FDC entries must be a multiple of 16:

$$F = \left\lceil \frac{\min(W \times C, E \times C) + 8}{16} \right\rceil \times 16 \quad [2]$$

Using equation 2, and constraining the values for the number of workspace IDs, *W*, and the number of Event Indexes, *E*, we can determine the number of FDC entries required for a B10 and S10 FDC in Single Core Mode. This analysis is performed in sections 2.3.2 and 2.3.3 below. First we perform an analysis for Dual Core Mode.

2.3.1.2 Dual Core Mode

For Dual Core Mode we split the Processor Cores into Protocol and Interface Cores. Each core is allocated a workspace ID. Note that in Dual Core Mode it is possible to have workspace IDs referenced from either an Event (from the Dispatcher) or an Inter-Core message. For both the B10 and S10 parts, the maximum number of Events per core is eight and the maximum number of outstanding Inter-Core messages is sixteen. Therefore, since there are only sixteen workspace IDs, in the worst case we will always run out of workspace IDs before Event Indexes or Inter-Core message slots. For this reason we do not consider the bounds on the FDC size due to Events or messages. We only consider the bounds due to the number of workspace IDs.

We begin the analysis with equation 3 that states that the number of Processor Cores is equal to the sum of the number of Protocol and Interface Cores.

$$C = C_p + C_i \quad [3]$$

The next two equations, 4 and 5, simply state that each Protocol and Interface Core can have no more than sixteen workspace IDs. This is a limitation of the hardware.

$$W_p \le 16 \quad [4]$$

$$W_i \le 16 \quad [5]$$

The total number of Interface Core workspace IDs divided by the number of Protocol Cores must equal the number of Protocol Core workspace IDs. The reason for this is that each Interface Core workspace ID requires a Protocol Core workspace ID to be allocated. This is expressed in equation 6 below. Note that we take the ceiling of the division.

$$W_p = \left\lceil \frac{W_i \times C_i}{C_p} \right\rceil \quad [6]$$

Combine equation 6 with equation 4 gives an equation for the number of Protocol Core workspace IDs:

$$W_p = \min\left(16, \left\lceil \frac{W_i \times C_i}{C_p} \right\rceil\right) \quad [7]$$

Combining equation 7 with equation 5 yields equation 8, a method for computing the number of Protocol Core workspace IDs given the number of Protocol Cores, C_p, and Interface Cores, C_i.

$$W_p = \min\left(16, \left\lceil \frac{16 \times C_i}{C_p} \right\rceil\right) \quad [8]$$

Similarly, the total number of Protocol Core workspace IDs divided by the number of Interface Cores must equal the number of Interface Core workspace IDs. The reason for this is that each Protocol Core workspace ID requires an Interface Core workspace ID to be allocated. Combining this with equation 5 gives equation 9 below:

$$W_i = \min\left(16, \left\lceil \frac{W_p \times C_p}{C_i} \right\rceil\right) \quad [9]$$

Combing equation 9 with equation 4 yields equation 10, a method for computing the number of Interface Core workspace IDs given the number of Protocol Cores, C_p, and Interface Cores, C_i.

$$W_i = \min\left(16, \left\lceil \frac{16 \times C_p}{C_i} \right\rceil\right) \quad [10]$$

The number of CAM entries that is required is equal to the minimum of the total number of Protocol Core workspace IDs and the total number of Interface Core workspace IDs. Note that we add eight to this number to allow for the LUC creating timer entries. This is for the same reason described in the Single Core Mode analysis above.

$$F' = \min(W_i \times C_i, W_p \times C_p) + 8 \quad [11]$$

Due to the hardware implementation, the number of FDC entries must be a multiple of 16:

$$F = \left\lceil \frac{\min(W_i \times C_i, W_p \times C_p) + 8}{16} \right\rceil \times 16 \quad [12]$$

If we fix the total number of Processor Cores, C, then using equations 3, 8, 10 and 12 we can calculate the number of FDC entries required. In the following sections we perform this calculation for the B10 and S10 parts.

2.3.2 B10 Flow Director CAM Size

The B10 part consists of two clusters with five processor cores per cluster. That gives C = 10. In Table 4 we display the ten different combinations of C_i and C_p. The case when C_i = 0 is Single Core Mode. From Table 4 we can see that the B10 FDC requires 96 entries.

Number Interface Cores, C_i	Number Protocol Cores, C_p	Number Interface Workspaces, W_i	Number Protocol Workspaces, W_p	Number FDC CAM Entries, F
1	9	16	2	32
2	8	16	4	48
3	7	16	7	64
4	6	16	11	80
5	5	16	16	96
6	4	11	16	80
7	3	7	16	64
8	2	4	16	48
9	1	2	16	32
0	10	0	16	96

Table 4: B10 Flow Director CAM Size

2.3.3 S10 Flow Director CAM Size

The S10 part consists of three clusters, again with five processor cores per cluster. That gives $C = 15$. In Table 5 we display the ten different combinations of C_i and C_p. The case when $C_i = 0$ is Single Core Mode. From Table 5 we can see that the B10 FDC requires 96 entries.

Number Interface	Number Protocol	Number Interface	Number Protocol	Number FDC

2. *Processor Core ID, Processor Workspace ID, Event Queue Element.* This is required for the processing of stateful events when the FDC is in single core mode. See section 2.4.5 for further details.
3. *Processor Core ID, Event Queue Element.* This is required for the processing of stateless events in either dual or single core mode. See section 2.4.6 for further details.
4. *Event Queue Element.* This is required for the processing of stateful events when an FDC entry is found. In that case the Core IDs and workspace IDs are already allocated – we simply need to allocate an event queue element. This is described in section 2.4.7.

Note that in all cases one and only one event queue element is allocated.

2.4.1 Event Queue Allocation

As described above, and event queue element is either allocated on a protocol core or an interface. So that it can forward the event, the Dispatcher must be informed as to whether a protocol core or interface core event queue element was allocated. When de-allocating the event queue element, the FDC must also be told whether this is a protocol core or interface core event. To facilitate this we use the concept of an *Event Mode*, as described in Table 6.

Event Mode Value	Description
00	Event Index does not describe a protocol core or interface core.
01	Event Index is for a protocol core.
10	Event Index is for an interface core.
11	Invalid.

Table 6: Event Mode Values

If the FDC is operating in single core mode then the Event Mode value should always be 01, i.e. it is the protocol core that uses the event index.

2.4.2 Event Type based Forwarding

The FDC manages the assignment and releasing of elements in the Processor Core's event queues and the Processor Core's workspace IDs. The Processor Core event queues are used to place packet, interface or timer events in. The Processor Core workspaces are used to place the actual flow state data in.

An extra dimension to this problem is that each Processor Core may process a certain protocol and not others, e.g. TCP may be on cores 0 through 15, but ICMP is only on cores 14 and 15. To solve this problem we use the *Event Type* as an index into an array of bitmaps where each bitmap represents which Processor Cores are capable of processing this event. Note that it is a Packet Processor in the IPU that assigns the 6-bit Event Type: the FDC simply has to understand them. To implement this feature the FDC keeps an array of masks. These masks are in Core Bitmap format, as described in section 2.1.3. Each mask is 15-bits wide and is used to indicate which Processor Cores are available for this type of event.

2.4.2.1 Event Type Sub-Structure

The Event Type also has sub-structure that allows the FDC to store 32 bitmaps instead of 64. This sub-structure is such that if the most significant bit (bit 5) of the Event Type is set, then the lower 5-bits contain a Core ID. This Core ID is turned into a Core Bitmap where only the bit for that specific core is set[6]. These event types are used to send messages to a specific core, and only that core. If the most significant bit (bit 5) of the Event Type is not set, then the lower 5-bits are used to index into an array of event masks that are held in FDC registers.

Note that when the MSB of the Event Type is set then by definition a stateless event is being processed[7]. We therefore know that if we are processing a stateful event then the MSB of the Event Type must be zero.

[6] If the resultant Core ID is not a valid value then a Core Bitmap of all zeros is used.
[7] For more information on stateless events the reader is directed to the Dispatcher HLD.

For this reason we do not use the MSB of the Event Type when indexing into the EVENT_MASK register for a stateful event. This is illustrated in sections 2.4.3, 2.4.4 and 2.4.7 below.

2.4.3 Variables for Processor Core Event Queue / Workspace ID Management

The following code[8] illustrates how event queue indexes and workspace IDs should be selected. Note that this code is used for illustration only, and is in no way related to the implementation.

First we define a bitmap called `event_bits` for each Processor Core, and a bitmap `workspace_id` for each Processor Core. If a bit is set then the corresponding event queue element or workspace ID is available for use. We also track the initial value of the `event_bits` register in `init_event_bits`. This allows us to determine when we have reached the end of the event queue.

The FDC must also keep the current write position, `head`, in the circular event queue. We also assume that the Processor Core keeps track of the current read position in the event queue.

Finally, we include the `pc_event_mask` and `ic_event_mask` registers that are indexed by the Event Type and provide a bitmap of which protocol or interface cores are available for processing this type of event.

```
reg [7:0] event_bits[14:0]          // Index is Core Index, result is bitmap of event elements
                                    // These are the FREEBUSY.EBITS bits (see 3.5.11)
reg [7:0] init_event_bits[14:0]     // The value that event_bits was initialised to
reg [15:0] workspace_id[14:0]       // Index is Core Index, result is bitmap of workspace IDs
reg [2:0] head[14:0]                // Index is Core Index, result is index of event queue head
reg [14:0] pc_event_mask[31:0]      // Index is event type, result is bitmap of Protocol Cores
                                    // These are the EVENT_MASK.PC_EVENT_MASK bits (see 3.5.12)
reg [14:0] ic_event_mask[31:0]      // Index is event type, result is bitmap of Interface Cores
                                    // These are the EVENT_MASK.IC_EVENT_MASK bits (see 3.5.12)
reg [1:0] event_mode                // Used to set the Event Mode value in an FDC response
```

The following are temporary registers that we use to perform the various allocation algorithms:

```
reg [14:0] ws_available             // Core Bitmap of processor cores that have a workspace available
reg [14:0] eq_available             // Core Bitmap of processor cores whose event queue head is free
reg [14:0] core_available_as_pc     // Core Bitmap of protocol cores that are available
reg [14:0] core_available_as_ic     // Core Bitmap of interface cores that are available
```

Given the above data structures, the following sections describe the rules for allocating and freeing event queue elements and workspace IDs.

2.4.4 Allocating a Protocol and Interface Core, an Event Queue Element, Two Workspace IDs

In this section we describe how to allocate a Processor Core and workspace ID pair. Two such Processor Cores are allocated, one for the protocol core and one for the interface core. This is used for the dual core mode of operation as determined by the CONTROL register of section 3.5.4. We assume that the allocation request has provided an Event Type, E. Based on that Event Type, and specifically the ALLOC_PC_EVENT bit of the EVENT_MASKS registers (see section 3.5.12), an Event Queue Element is also allocated on one of the cores.

This method of allocation requires an entry in the FDC to be created. If there is no space in the FDC then other resources must not be allocated. Note that the number of available workspaces and Event indexes is completely independent of the number of FDC entries, i.e. we may run out of workspaces and event indexes before FDC entries, or vice-versa. For this reason there must be a mechanism for either detecting a full FDC, or detecting that an FDC entry could not be created due to no space.

[8] This is pseudo Verilog and is for descriptive purposes only.

The first step we take is to build the `ws_available` and `eq_available` Core Bitmaps. We do this by examining each processor core in a for loop, and setting the `ws_available` and `eq_available` bit appropriately. The `ws_available` bit is set for a core if any workspace is available. The `eq_available` bit is set only if the head of the event queue is free.

```
for (i = 0; i < MAX_CORE_BITMAP_NUM; i = i + 1)
        ws_available[i] = (workspace_id[i][0] | workspace_id[i][1] | .. | workspace_id[i][15]);
        eq_available[i] = event_bits[i][head[i]];
```

Now we compute the `core_available_as_pc` and `core_available_as_ic` bitmaps by combining the `ws_available` bitmap with the appropriate mask[9]. Note that we since this is stateful processing we only use the lower 5 bits of the Event Type, as described in section 2.4.2.1.

```
core_available_as_pc = ws_available & pc_event_mask[E & 5'h1f];
core_available_as_ic = ws_available & ic_event_mask[E & 5'h1f];
```

Finally, we allow for the allocation of an Event Queue element. Note that only one core will need an event queue element to be allocated. Which core to allocate an event queue element for is determined by the ALLOC_PC_EVENT bit of the EVENT_MASK register (see section 3.5.12).

```
if (ALLOC_PC_EVENT[E & 5'h1f])
        core_available_as_pc = core_available_as_pc & eq_available;
else
        core_available_as_ic = core_available_as_ic & eq_available;
```

We now wish to select a protocol core and an interface core from the respective available bitmaps. However, we want to select them such that the protocol core and interface core are on the same cluster. The way we do this is to mask out all protocol cores that do not have an interface core available in their cluster.

```
cluster_0_has_ic = core_available_as_ic[0] | core_available_as_ic[1] | core_available_as_ic[2]
                 | core_available_as_ic[3] | core_available_as_ic[4];
cluster_1_has_ic = core_available_as_ic[5] | core_available_as_ic[6] | core_available_as_ic[7]
                 | core_available_as_ic[8] | core_available_as_ic[9];
cluster_2_has_ic = core_available_as_ic[10] | core_available_as_ic[11] | core_available_as_ic[12]
                 | core_available_as_ic[13] | core_available_as_ic[14];
cluster_0_has_pc = core_available_as_pc[0] | core_available_as_pc[1] | core_available_as_pc[2]
                 | core_available_as_pc[3] | core_available_as_pc[4];
cluster_1_has_pc = core_available_as_pc[5] | core_available_as_pc[6] | core_available_as_pc[7]
                 | core_available_as_pc[8] | core_available_as_pc[9];
cluster_2_has_pc = core_available_as_pc[10] | core_available_as_pc[11] | core_available_as_pc[12]
                 | core_available_as_pc[13] | core_available_as_pc[14];
cluster_0_has_ic_and_pc = cluster_0_has_ic & cluster_0_has_pc;
cluster_1_has_ic_and_pc = cluster_1_has_ic & cluster_1_has_pc;
cluster_2_has_ic_and_pc = cluster_2_has_ic & cluster_2_has_pc;
```

We now select from `cluster_0_has_ic_and_pc`, `cluster_1_has_ic_and_pc` and `cluster_2_has_ic_and_pc` in a round robin fashion. This effectively balances the selection of the Processor Core across the clusters, which is a requirement.

Based on which cluster was selected, we then select a Protocol Core by performing round robin on either `core_available_as_pc`[0:4], `core_available_as_pc`[5:9] or `core_available_as_pc`[10:14]. This balances the load across the Protocol Cores within a cluster. Suppose that bit p (Core Index p) is selected from the `core_available_as_pc` Core Bitmap. Using Table 1 we then convert this Core Index value into a protocol core ID, P. We must now select an interface core in that protocol cores cluster. To do this we use the same technique as we did to select the Protocol Core, i.e. based on which cluster was selected, we select an

[9] For the same Event Type, the software must ensure that `ic_event_mask` and `pc_event_mask` do not overlap. See section 3.5.12 for further details.

Interface Core by performing round robin on either `core_available_as_ic`[0:4], `core_available_as_ic`[5:9] or `core_available_as_ic`[10:14]. This balances the load across the Interface Cores within a cluster.

Let us suppose that bit i (Core Index i) is selected. Using Table 1 we then convert this Core Index into an interface Core ID, I. We then select a workspace ID, `chosen_pc_ws_id`, from `workspace_id[p]` – we can use any available workspace ID, e.g. first bit set. Similarly, we select a workspace ID, `chosen_ic_ws_id`, from `workspace_id[i]`.

We must now de-allocate the resources that have just been consumed. Depending on which event queue element is allocated, we increment the appropriate processor cores `head` value, allowing it to wrap at 8 (5'h7 mask). We must also allow for when not all 8 event queue elements are in use. We do this by checking the value of the `init_event_bits` variable at the head – if it is zero then we have moved beyond where the event queue length was initialised, and we must wrap the head back to zero[10].

```
if (ALLOC_PC_EVENT[E & 5'h1f])
        event_mode = 2'b01;
        event_bits[p][head[p]] = 0;head[p] = (head[p] + 1) & 5'h7;
        if init_event_bits[p][head[p]] is 0 then
                head[p] = 0;
else
        event_mode = 2'b10;
        event_bits[i][head[i]] = 0;
        head[i] = (head[i] + 1) & 5'h7;
        if init_event_bits[i][head[i]] is 0 then
                head[i] = 0;
workspace_id[p][chosen_pc_ws_id] = 0;
workspace_id[i][chosen_ic_ws_id] = 0;
```

2.4.5 Allocating a Processor Core, Event Queue Element, Workspace ID

This method of allocation is used when the FDC is in single core mode and we wish to allocate a processor core and workspace ID pair for stateful event processing. The mode of the FDC is determined by the CONTROL register of section 3.5.4. Note that due to the definition of the ALLOC_PC_EVENT bit (see section 3.5.12), we know that the software always sets this bit to 1 in Single Core Mode, therefore ensuring that the PC_EVENT_MASK is used.

The algorithm that is used to select a Processor Core, Event Queue Element and Workspace ID can be exactly the same as that of section 2.4.4 above, with a few of the steps overridden:

1. The `core_available_as_ic` should be forced to all ones just after it is assigned the value `ws_available & ic_event_mask[E & 5'h1f]`. This effectively marks all interfaces cores as available, allowing any available protocol core to be selected.
2. No interface core or workspace ID should be allocated.

This method of allocation requires an entry in the FDC to be created. If there is no space in the FDC then other resources must not be allocated. As was described in section 2.4.4, the number of available workspaces and Event indexes is completely independent of the number of FDC entries.

2.4.6 Allocating a Processor Core, Event Queue Element, no Workspace ID

This is when we just want to allocate an event queue element for a Processor Core, and we do not wish to allocate a workspace ID. An example of this occurs when an ARP frame is received: we need to pass an event to one or more Processor Cores, but there is no associated flow state. This method of allocation does not create a new FDC entry, so allocation still continues even if the FDC is full.

[10] This assumes that when the event queues free/busy bits are initialised on the FDC, event queue elements must be used from the bottom up and cannot contain holes. See section 3.5.11 for more details.

There are two variants of this type of operation. In the first method of operation the FDC uses a supplied Event Type to select an Event Mask. This Event Mask will restrict the selection of the Processor Core to a subset of the available Processor Cores. In the second method of operation the Event Mask is supplied to the FDC directly. Note that in either case these masks are in Core Bitmap format, as described in section 2.1.3.

To allocate a processor core, we first determine the correct event mask to use. If a mask was supplied then we use it directly. If a mask was not supplied then we determine the event mask allowing for the Event Type sub-structure as defined in section 2.4.2.1. Note that when the event mask is not supplied, and the MSB of the Event Type is not set, we use the logical or of the values of the PC_EVENT_MASK and IC_EVENT_MASK from the EVENT_MASK register (see section 3.5.12)[11]. A Processor Core, Core Index n, is therefore available if:

```
if event_mask is supplied then
        local_event_mask = Mask from the FDC command;
else if E has MSB set then
        local_event_mask = Core Bitmap formed using lower 5-bits of E as a Core ID;
else
        local_event_mask = pc_event_mask[E & 5'h1f] | ic_event_mask[E & 5'h1f];
core_available = event_bits[n][head[n]] & local_event_mask;
```

We can now compute for all cores which ones are available. Out of all of these available cores we select one based on a round robin algorithm, i.e. we want to distribute the load such that repeated requests to allocate an event queue element go to different Protocol Clusters, and then different Processor Cores within that cluster.

If a core has been selected then we allocate an event queue element. Suppose that Core Index p is selected:

```
event_bits[p][head[p]] = 0;
head[p] = (head[p] + 1) & 5'h7;
if (init_event_bits[p][head[p]] == 0)
        head[p] = 0;
```

If the event is not available then we indicate an "out of space" response to the device requesting an Event Queue element. It can then try again later in the hope that there will be space in the FDC.

2.4.7 Allocating an Event Queue Element, no Workspace ID

This method of allocation is when an FDC entry is found for a stateful event. In that case we need to allocate an event queue element so that the event can be forwarded to a processor core, but we do not need to allocate a workspace ID since one has already been allocated. This method of allocation does not create a new FDC entry, so allocation still continues even if the FDC is full.

The first step in allocating an event queue element is to determine which processor core to send it to: the protocol core or the interface core. This is done using the supplied Event Type, E, and the ALLOC_PC_EVENT bit of the EVENT_MASK register of section 3.5.12. Let us assume that the FDC entry has Protocol Core ID value P and Interface Core ID I, which is Core Index values p and I respectively. We allocated an Event Index as follows:

```
if (ALLOC_PC_EVENT[E & 5'h1f])
        event_mode = 2'b01;
        event_bits[p][head[p]] = 0;
        head[p] = (head[p] + 1) & 5'h7;
        if init_event_bits[p][head[p]] is 0 then
                head[p] = 0;
else
```

[11] Using the logical or of the two event masks means that both protocol and interface cores can be selected for stateless event processing.

```
event_mode = 2'b10;
event_bits[i][head[i]] = 0;
head[i] = (head[i] + 1) & 5'h7;
if init_event_bits[i][head[i]] is 0 then
        head[i] = 0;
```

Note in the above how since this is a stateful allocation, we only examine the lower 5 bits of the Event Type (see section 2.4.2.1). The above pseudo code follows the exact same steps as those described at the end of section 2.4.4, and the reader is directed there for further explanation.

2.4.8 Releasing an Event Queue

The FDC can be requested to release an event queue element via two methods. In the first method the FDC is requested to release an event queue element that is associated with an FDC entry. In this case the FDC is supplied with an *Event Mode* value as described in section 2.4.1. The FDC uses this to determine whether it should free an event queue element for the protocol core, the interface core, or neither. Suppose that we are releasing an Event Number[12] Q and that the FDC entry has Protocol Core value P (Core Index p) and Interface Core value I (Core Index i):

```
if (event_mode[0])
        event_bits[p][Q] = 1;
else if (event_mode[1])
        event_bits[i][Q] = 1;
```

In the second method the FDC can be instructed to release an event queue element that is not associated with an FDC entry. In this case the FDC command will supply the processor core from which the event queue element is to be released. If the supplied processor core value is P (Core Index p), and event element Q is being released, then the release is simply:

```
event_bits[p][Q] = 1;
```

2.4.9 Release a Workspace ID

To release a workspace ID W for a Processor Core P (Core Index p) we simply mark that workspace ID as available:

```
workspace_id[p][W] = 1;
```

2.4.10 Workspace Sizes

The size of a workspace is configurable and can range from 128 to 2K bytes. The Protocol Cluster allows the workspace area of its dual port memory to be addressed by an index ranging from 0 through 15, where each Index represents a 128-byte block of memory. If the workspace is configured to be 128 bytes then the addressing scheme is obvious: a workspace ID maps directly to a Protocol Cluster workspace index.

How are workspace sizes other than 128 bytes supported? If the workspace size is 512 bytes then the ACP software should initialise the FDC such that workspaces IDs 0, 4, 8 and 12 are available in the WBITS section of the FREEBUSY register (section 3.5.11). Workspace IDs 1, 2, 3, 5, 6, 7, 9, 10, 11, 13, 14 and 15 should be marked as unavailable. Using this scheme we will then write 512 byte workspaces to Protocol Cluster workspace indexes 0, 4, 8 and 12, which is as required. Similarly, for 1K byte workspaces the WSBITS should be initialised with workspace IDs 0 and 8 available. For 2K byte workspaces on workspace ID 0 should be marked as available.

2.4.11 Event Queue Length

The size of an event is fixed at 256 bytes maximum. Events may be smaller than this, but for the purposes of queue space allocation we can assume that they are never larger than 256 bytes. One thing that can vary

[12] Shifting the event index right by one forms the event number.

on the Processor Cores is how deep the event queue is, i.e. how many Event Numbers are there. This is programmed into the FDC by setting the EBITS fields of the FREEBUSY register (section 3.5.11) correctly.

For example, if the event queue is eight elements deep, then EBITS should be initialised to value 8'b11111111. However, if the event queue is only four elements deep then EBITS should be initialised to 8'b00001111. Note that the event numbers assigned **must** start from zero otherwise the FDC cannot function correctly, i.e. 8'b00110011 is not valid. Also note that these are event numbers, as opposed to event indexes. See section 2.1.4 for clarification.

2.4.12 Resource Allocation Implementation Guide

In the previous sections we discussed how Processor Cores and workspace IDs can be selected assuming the presence of a round robin algorithm. In this section we describe how a simple round robin scheme could be implemented with a fixed (small) number of clocks.

Figure 4 illustrates the components of this algorithm. The input to the algorithm is an *Available Bitmap* that describes which objects are available for this iteration of the round robin algorithm. For example, this value could be the `core_available_as_pc` bitmap of section 2.4.4.

We then use a *Round Robin* register that is initialised to all ones at the start of day. This is logically AND'ed with the *Available Bitmap* register to create the *Round Robin Available Bitmap* register. If the *Round Robin Available Bitmap* register is not zero then we select an object (bit) using a simple first bit set operation[13]. To prevent this bit from being selected in the next round robin we clear the corresponding bit in the *Round Robin* register. If the *Round Robin Available Bitmap* register is zero then we select an object using a first bit set operation on the *Available Bitmap* register. We then reset the *Round Robin* register to all ones, with the currently selected bit set to zero.

Note that this is no need to explicitly check the *Round Robin* register for value zero and to reset it. This automatically drops out in the case when the *Round Robin Available Bitmap* register is zero.

It can be seen that this logic approximates a round robin selection algorithm. However, to the external observer the algorithm may not appear to be *exactly* round robin. The reason is that it will always pick the lowest numbered bit, so if a bit becomes available then it may be selected, even though it is a lower numbered bit than one previously selected.

[13] A description of the first bit set logic is not included, but it can be performed in a very small number of clocks, perhaps even one clock given the small number of inputs.

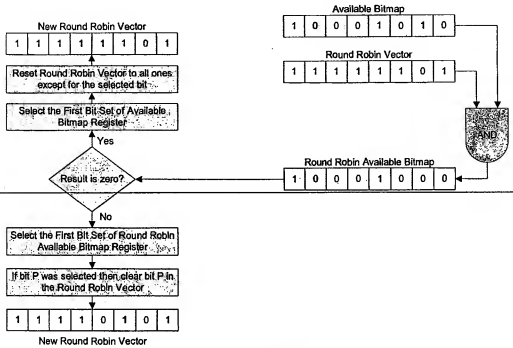

Figure 4: Round Robin Selection Logic

The FDC will need to record at least two different *Round Robin Available Bitmaps*: one for the protocol core selection and one for the interface core selection[14]. Note that we do not keep a *Round Robin Available Bitmap* per Event Type, i.e. we perform global round robin across all Event Types.

2.5 FDC Commands

2.5.1 FDC Requests

Table 7 illustrates the commands that can be issued to the FDC by the Dispatcher and the LUC. The use of these commands will be explained in more detail in the following sections..

Command	Originator	Description	Purpose
CRTIMER	LUC	Create timer entry.	Used by the LUC when a timer event occurs. The intent is to create an entry in the FDC so that the timer can be serviced.
GETEVENT	Dispatcher	Get Event Number on a Processor Core. This will be used for frames that do not require a workspace, e.g. ARP.	Used by the Dispatcher to get an available Event Number for a specific Processor Core. This is used for non-TCP frames, and allows the Dispatcher to place events into a Processor Cores event queue.

[14] In the FDC RTL implementation, in dual core mode the *Round Robin Vector* for an Interface Core is always adjusted even if an Interface Core is not allocated (GETEVENT), i.e. when we clear the bit in the Protocol Core *Round Robin Vector* we clear the same bit in the Interface Core *Round Robin Vector*. This helps prevent an Interface Core from receiving an un-balanced number of stateless and stateful events.

Command	Originator	Description	Purpose
LFKCREATE	Dispatcher	Lookup with Flow Key and Create.	Used by the Dispatcher to lookup a packet or interface event flow key. If an FDC entry is not found then one is created.
RELEVENT	Dispatcher	Release an Event Number on a Processor Core.	Used by the Dispatcher to release an Event Queue element that was claimed using GETEVENT. The Dispatcher knows to use this command by virtue of a field in the "Done Event" from the Processor Core.
RMFIDX	LUC	Remove with FDC Index.	Used by the LUC to remove an entry from the FDC. Note that we use an FDC index, and not a flow key.
SERVTIMER	Dispatcher	Service a timer.	Used by the Dispatcher when it wants to service a timer event that the LUC created in the FDC.
TDFIDX	Dispatcher	Tear down with FDC Index.	Used by the Dispatcher to indicate that a flow is being torn down by the LUC.
UPFIDX	Dispatcher	Update with FDC Index.	Used by the Dispatcher to indicate that a flow is being updated by the LUC.

Table 7: FDC Commands

2.5.2 FDC Responses and Errors

All FDC commands require a response to be issued to the requestor. Always providing a response, even if that response is an empty response, simplifies the design of the requestor. If the FDC request also required an interaction with the FDC state machine[15] then we must also be capable of indicating the following:

- Command not valid in this state. This error should never occur and is considered fatal. If this condition does occur then either the FDC or one of its requestors is not working to specification.
- No space left in FDC CAM or no Processor Cores event queues / workspaces available. This condition is expected to occur under stress conditions.

Rather than defined new fields in the response to indicate these conditions, we overload the Flags field[16]. We use the value 5'b11111 to indicate a fatal error, and 5'b11110 to indicate no space available. If the Flags field of the FDC response is neither of these values then it is indicating an FDC entry state using the encoding of Figure 5.

2.6 Flow Director State Diagram and Transitions

Figure 5 shows the state transition diagram for a **single element** of the FDC. Each node in this diagram corresponds to a different state, with the VPTDZ terminology indicating the Valid, Pending, Timer, Deleting and Zero flags respectively. The initial state is the *CHECKED IN* state, where the entry is not valid, i.e. there is no entry in the FDC.

It should be noted that Figure 5 does not include any transitions for the GETEVENT or RELEVENT commands. The reason is that these commands do not cause state transitions for FDC entries: they simply allocate and de-allocate event queue / Processor Cores.

[15] All FDC commands except GETEVENT require this state machine interaction.
[16] This is overloading the Flags field of the FDC response only, and not the FDC entry format.

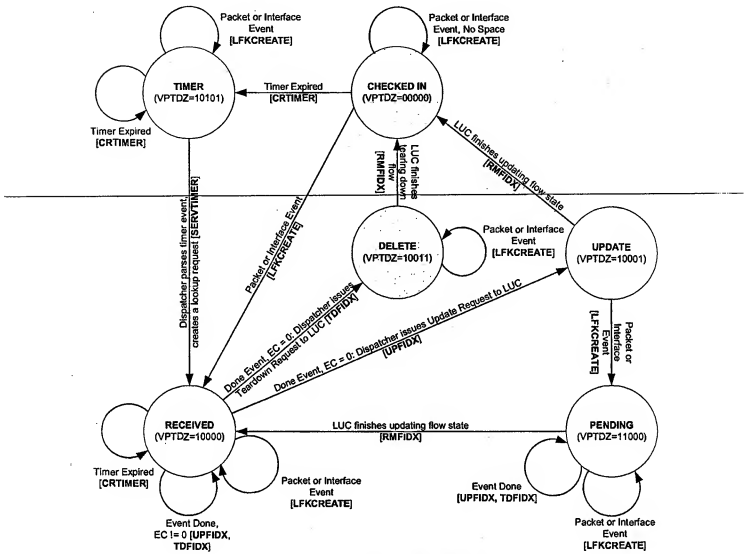

Figure 5: Flow Director State Diagram

It should be noted that only the *CHECKED IN*, *TIMER*, *RECEIVED* and *DELETE* states require transitions for the *Timer Expired* event. The reason is that the LUC HLD specifies that it can only issue timer events for flows that are not in its Timer Cache. For the *UPDATE* and *PENDING* states the flow will definitely be in the LUCs Timer Cache[17]. It would also seem that the flow should be in the Timer Cache during the DELETE state, but due to LUC implementation details the flow is temporarily deleted from the Timer Cache. See section 2.6.6 for more details.

Similarly, the *Checked In*, *Delete* and *Timer* states do not require a transition for *Event Done* events, since an event done message can only be issued after a Processor Core has finished processing a frame/interface

[17] Since the LUC can only issue timer events for flows that are in the CHECKED IN state, it is possible that a constant stream of frames can prevent a timeout from occurring. Having the processor cores keep track of time for each workspace can solve this condition. See the TCP/IP Software HLD for further details.

event: this is not possible since in these states there can never be any outstanding events on the Processor Cores.

2.6.1 Flow Chart Usage

In the sections 2.6.3 through 2.6.8 we describe each state in more detail and provide an action/response table for all commands that can be received in that state. These sections should be interpreted as follows:

1. When a command is received use the supplied Flow Key or FDC Index to find the associated entry in the CAM. If the entry is not present then the valid bit is zero, i.e. we are in the *CHECKED IN* state.
2. Now that we know the state of the entry, find the appropriate section below.
3. Find the command in the table and use the *Action* and *Response and Next State* columns to determine what needs to be done, what the next state should be for that FDC entry, and what the response from the FDC should be.

2.6.2 FDC Command Usage and Assumptions

In order to guarantee correct operation of the FDC, some rules must be followed regarding the FDC usage:

1. Once a TDFIDX command has been issued for a flow, all future commands must be TDFIDX, i.e. you cannot ask for a flow tear down, and then follow it with a flow update. An example might be getting a frame after a RST has been received. In this case the Processor Cores must mark the workspace (which remains valid throughout this critical section) as "torn down" and respond with the appropriate TDFIDX command. Note that when the FDC receives these TDFIDX commands it will do nothing until the Event Count reaches zero, at which point the Dispatcher will issue a tear down command to the LUC.
2. Once an FDC entry enters the *TIMER* state all LFKCREATE commands are ignored. This implies that the Dispatcher should back off and service the timer for that flow (SERVTIMER).
3. If the response of an LFKCREATE command indicates the *DELETE* state then the packet or interface event will not be forwarded to a processor core. The Dispatcher must back off and re-issue the command at a later date. If the Dispatcher were to forward events after finding the FDC in the DELETE state then those events would not be tracked in the event count, effectively causing the FDC to operate incorrectly.
4. If the response of an UPFIDX or TDFIDX command indicates the *PENDING* state then the Dispatcher must back off and re-issue the command at a later date. This ensures that the LUC only receives an update command once, and that the LUC update command is allowed to completely finish before any new updates are issued.
5. It is assumed that when the LUC has finished updating a flow state, but before it issues the RMFIDX command, it clears the valid bit in the workspace. If after issuing the RMFIDX command the LUC sees that the state is *PENDING* then the LUC must set the valid bit again.

2.6.3 Checked In State

Indicated by: VPTDZ = 00000

Each entry of the FDC starts in the *CHECKED IN* state, where the Valid bit is not set, i.e. the entry is not valid and is unused.

Transitions out of this state indicate that a CAM entry is being created, while transitions into the state indicate that a CAM entry has been deleted.

The first transition out of this state occurs when the Dispatcher receives a packet or interface event and issues a lookup command to the FDC (LFKCREATE). In this case the FDC creates an entry, and puts it into the *RECEVIED* state. The second transition out of this state occurs when the LUC determines that a timer has expired for a flow and issues a create timer (CRTIMER) command to the FDC. In this case the FDC examines the SMC_DISP_Almost_Full signal and, if allowed to, creates an entry and puts it into the *TIMER* state[18]. See section 2.7 for further information on the SMC_DISP_Almost_Full signal.

Table 8 illustrates the action and response for each command that can be received in the *CHECKED IN* state. The *CHECKED IN* state basically means that the entry was not found.

Command	Originator	Description	Action	Response and Next State
CRTIMER	LUC	LUC wants to create an FDC entry due to an expired timer.	Create an entry in the FDC. Do not allocate any Processor Cores,	*Entry Created:* FDC Index = FDC Index of new entry Response Flags = TIMER
		Since we are in the *CHECKED IN* state we know that the entry does not currently exist.	workspace IDs or an event number. Give the Event Count value zero. Note that the SMC_DISP_Almost_Full signal must be examined, as described in section 2.7.	Next State = TIMER *Entry Not Created due to either lack of space or SMC_DISP_Almost_Full being set with reg[CONTROL].DIS_SMC_DISP_BP clear.* Flags = 5'b11110 Next State = CHECKED IN
LFKCREATE	Dispatcher	Dispatcher received a packet or interface event. Since we are in the *CHECKED IN* state we know that the entry does not currently exist.	Create an entry in the FDC. Allocate Processor Cores, workspace IDs and an event number according to section 2.4.4. The Event Count is set to value one if an FDC entry is successfully created.	*Entry Created:* FDC Index = FDC Index of new entry Event Index = Allocated event number << 1 Protocol Core ID = Allocated protocol core Protocol Workspace ID = Allocated workspace Interface Core ID = Allocated interface core Interface Workspace ID = Allocated workspace Event Mode = Value according to section 2.4.4 New Entry = 1 Response Flags = RECEIVED Next State = RECEIVED *Entry Not Created due to lack of space or unavailability of Processor Core(s) / workspace(s) / event number.* Response Flags = 5'b11110 Next State = CHECKED IN
RMFIDX	LUC	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = CHECKED IN
SERVTIMER	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = CHECKED IN
TDFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = CHECKED IN
UPFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = CHECKED IN

Table 8: *Check In* State Transitions

[18] The DIS_SMC_DISP_BP bit of the CONTROL register can be used to override the SMC_DISP_Almost_Full signal.

2.6.4 Received State

Indicated by: VPTDZ = 10000

The *RECEIVED* state indicates that this FDC entry is valid, and that we are currently working on an event for this flow.

The first transition out of this state occurs when the Dispatcher receives an Event Done message and issues an updated (UPFIDX) or tear down (TDFIDX) command. This indicates that a *Processor Core* has finished processing an event that was issued to it. In this case the Dispatcher will decrement the Event Count (EC) of the FDC entry. If this Event Count reaches zero then the FDC entry is transitioned into the *UPDATE* or *DELETE* state dependent upon whether an UPFIDX or TDFIDX command was received, and the Dispatcher issues an update or tear down message to the LUC. If the Event Count does not reach zero then the FDC is left in the *RECEIVED* state.

Another transition out of the RECEIVED state occurs when the FDC receives a lookup command (LFKCREATE) due to the Dispatcher receiving another packet or Interface event. In this case the FDC entry is found and the Event Count is incremented, leaving the FDC entry in the RECEIVED state.

Command	Originator	Description	Action	Response and Next State
CRTIMER	LUC	LUC is trying to create an expired timer entry, but a LUC request is in transit for this flow.	No action required.	Response Flags = RECEIVED Next State = RECEIVED
LFKCREATE	Dispatcher	Dispatcher received a packet or interface event.	If the Event Count has value 7'h7f then set the EC_OVFLOW flag in the STATUS register. Note that processing continues as normal regardless of whether the EC_OVFLOW bit was set. Increment Event Count. Allocate an Event Number according to section 2.4.7 using the appropriate Processor Core of the found entry.	*Event Number was available on that Processor Core:* FDC Index = FDC Index of entry Event Index = Allocated Event Number << 1 Protocol Core ID = Protocol Core ID of entry Protocol Workspace ID = Workspace of entry New Entry = 0 Interface Core ID = Interface Core ID of entry Interface Workspace ID = Workspace of entry Event Mode = Value according to section 2.4.7 Response Flags = RECEIVED Next State = RECEIVED *No Event Number was available:* Response Flags = 5'b11110 Next State = RECEIVED
RMFIDX	LUC	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = RECEIVED
SERVTIMER	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = RECEIVED

Command	Originator	Description	Action	Response and Next State
TDFIDX	Dispatcher	Dispatcher received a tear down message from Processor Core.	Decrement Event Count. De-allocate an Event Number according to section 2.4.8 using the Event Index and Event Mode of the TDFIDX command and the appropriate Processor Core of the FDC entry. Note that once a TDFIDX command has been issued, all future	If new Event Count is zero Response Flags = DELETE Next State = DELETE else Response Flags = RECEIVED Next State = RECEIVED
			Dispatcher commands for this flow will be TDFIDX (see section 2.6.2).	
UPFIDX	Dispatcher	Dispatcher received a done message from Processor Core.	Decrement Event Count. De-allocate an Event Number according to section 2.4.8 using the Event Index and Event Mode of the UPFIDX command and the appropriate Processor Core of the FDC entry.	If new Event Count is zero Response Flags = UPDATE Next State = UPDATE else Response Flags = RECEIVED Next State = RECEIVED

Table 9: *Received* State Transitions

2.6.5 Update State

Indicated by: VPTDZ = 10001

The *UPDATE* state is entered just after the Dispatcher has requested that the LUC update its flow state memory with the new flow state.

A transition out of the *UPDATE* state occurs when the LUC finishes updating the flow state. In this case the LUC will issue a remove (RMFIDX) command to the FDC to delete the entry, putting it back to the *CHECKED IN* state. Note that before the LUC issues the RMFIDX command it must set both workspaces to invalid: this ensures that the next flow that uses these workspace IDs does not incorrectly start processing it before it is ready.

Another transition out of the *UPDATE* state occurs when the Dispatcher receives a packet or interface event and issues a lookup command to the FDC (LFKCREATE). Such an event causes the FDC entry to enter the *PENDING* state. There is no need to copy either workspace contents to either of the Processor Cores: the old versions that are present are still correct, but they will not be marked as valid yet[19].

[19] When the LUC issues a RMFIDX command and sees that an entry is in the PENDING state it will go back to the workspaces and mark them as valid.

Command	Originator	Description	Action	Response and Next State
CRTIMER	LUC	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = UPDATE
LFKCREATE	Dispatcher	Dispatcher received a packet or interface event.	Increment Event Count. Allocate an Event Number according to section 2.4.7 using the appropriate Processor Core of the found entry.	*Event Number was available on that Processor Core*: FDC Index = FDC Index of entry Event Index = Allocated Event Number << 1 Protocol Core ID = Protocol Core ID of entry Protocol Workspace ID = Workspace of entry Interface Core ID = Interface Core ID of entry Interface Workspace ID = Workspace of entry Event Mode = Value according to section 2.4.7 New Entry = 0 ~~Response Flags = PENDING~~ Next State = PENDING *No Event Number was available*: Response Flags = 5'b11110 Next State = UPDATE
RMFIDX	LUC	LUC finished updating the flow state	De-allocate the Protocol and Interface Core's workspace IDs according to section2.4.9.	Response Flags = CHECKED IN Next State = CHECKED IN
SERVTIMER	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = UPDATE
TDFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = UPDATE.
UPFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = UPDATE

Table 10: *Update* State Transitions

2.6.6 Delete State

Indicated by: VPTDZ = 10011

The *DELETE* state is entered just after the Dispatcher has requested that the LUC teardown a flow whose Event Count was found to be zero. This state is used to indicate that any events received during the tear down process should be either discarded or sent to a Processor Core as "flow is being torn down".

A transition out of the *DELETE* state can only occur when the LUC has finished deleting the flow and removes the FDC entry with the RMFIDX command.

Note that the CRTIMER command is not valid in the *DELETE* state. The LUC guarantees not to issue a CRTIMER while the FDC entry is in this state. See the LUC HLD for further details.

Command	Originator	Description	Action	Response and Next State
CRTIMER	LUC	Invalid command	Set invalid command bit in status register.	Response Flags = 5'b11111 Next State = DELETE
LFKCREATE	Dispatcher	Dispatcher received a packet or interface event.	No action required.	Response Flags = DELETE Next State = DELETE
RMFIDX	LUC	LUC finished updating the flow state	De-allocate the Protocol and Interface Core's workspace IDs according to section2.4.9.	Response Flags = CHECKED IN Next State = CHECKED IN
SERVTIMER	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = DELETE
TDFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = DELETE
UPFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = DELETE

Table 11: *Delete* State Transitions

2.6.7 Pending State

Indicated by: VPTDZ = 11000

The *PENDING* state is used to indicate that the Dispatcher requested that the LUC update its flow state memory, but then another packet or application event arrived. In this case the Dispatcher will pass the packet or application event to the Processor Cores to be processed. The Processor Cores are then allowed to process this event, since the workspace IDs are still marked as valid[20]. When the LUC issues the RMFIDX command the FDC transforms the state of the entry to the *RECEIVED*, just as if a packet or interface event had arrived when an Update request was not pending.

If more lookup request commands (LFKCREATE) are received while in the *PENDING* state then we simply increment the Event Count (EC) and remain in the *PENDING* state.

We must also allow for receiving done messages from the Processor Cores with the FDC entry in the *PENDING* state. In this state the FDC does nothing in response to UPFIDX or TDFIDX commands: it just indicates that the FDC entry is in the *PENDING* state. This assumes that the Dispatcher will back off and re-issue the FDC command at a later date. This is where we ensure that the LUC does not have two updates active for the same flow.

[20] There may be a brief period when the workspaces are marked as invalid: in between the time the LUC marks it as invalid, issues the RMFIDX command, sees the entry is in the *PENDING* state, and then marks the workspaces as valid again.

Command	Originator	Description	Action	Response and Next State
CRTIMER	LUC	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = PENDING
LFKCREATE	Dispatcher	Dispatcher received a packet or interface event.	If the Event Count has value 7'h7f then set the EC_OVFLOW flag in the STATUS register. Note that processing continues as normal regardless of whether the EC_OVFLOW bit was set. Increment Event Count. Allocate an Event Number according to section 2.4.7 using the appropriate Processor Core of the found entry.	*Event Number was available on that Processor Core*: FDC Index = FDC Index of entry Event Index = Allocated Event Number << 1 Protocol Core ID = Protocol Core ID of entry Protocol Workspace ID = Workspace of entry Interface Core ID = Interface Core ID of entry Interface Workspace ID = Workspace of entry Event Mode = Value according to section 2.4.7 New Entry = 0 Response Flags = PENDING Next State = PENDING *No Event Number was available*: Response Flags = 5'b11110 Next State = PENDING
RMFIDX	LUC	LUC finished updating the flow state	No action required.	Response Flags = RECEIVED Next State = RECEIVED
SERVTIMER	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = PENDING
TDFIDX	Dispatcher	Dispatcher received a done with tear down event from a Processor Core.	No action required.	Response Flags = PENDING Next State = PENDING
UPFIDX	Dispatcher	Dispatcher received a done with update event from a Processor Core.	No action required.	Response Flags = PENDING Next State = PENDING

Table 12: *Pending* State Transitions

2.6.8 Timer State

Indicated by: VPTDZ = 10101

The *TIMER* state is used to indicate that the LUC has a timer pending on a particular flow. This is the only case when the LUC will create an entry in the FDC. The only way to transition out of the *TIMER* state is for the FDC to receive a service timer (SERVTIMER) command. This will occur when the Dispatcher places a lookup request to the LUC.

On receipt of a SERVTIMER command the FDC can transition the entry to the *RECEIVED* state. Note that during the *TIMER* state it is not possible to have packet or interface events – the Dispatcher must service the timer event before it services any packet or interface events.

On receipt of a CRTIMER command the FDC remains in the *TIMER* state and does not indicate an error. This allows the LUC to issue a CRTIMER command even if it has already successfully done so. The only reason the LUC would do this is if it simplified its implementation. The LUC could just as well determine that it has already issued the CRTIMER, and not issue another one.

Command	Originator	Description	Action	Response and Next State
CRTIMER	LUC	LUC is trying to create an expired timer entry.	No action required.	Response Flags = TIMER Next State = TIMER
LFKCREATE	Dispatcher	Dispatcher received a packet or interface event.	None. The Dispatcher will see that Flags = TIMER in the FDC response, and it must then service the Timer Event Queue before it can re-try this Packet / Interface Event.	Flags = TIMER Next State = TIMER
RMFIDX	LUC	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = TIMER
SERVTIMER	Dispatcher	Dispatcher is servicing the timer event	Increment Event Count. Allocate Processor Cores, workspace IDs and Event Number according to section 2.4.4. Note that by definition the Event Count of an entry in the *TIMER* state must be zero, so there is no need to check for overflow.	*Event Number was available on that Processor Core*: FDC Index = FDC Index of entry Event Index = Allocated Event Number << 1 Protocol Core ID = Allocated protocol core Protocol Workspace ID = Allocated workspace Interface Core ID = Allocated interface core Interface Workspace ID = Allocated workspace Event Mode = Value according to section 2.4.4 Response Flags = RECEIVED Next State = RECEIVED *No Event Number was available*: Flags = 5'b11110 Next State = TIMER
TDFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = TIMER
UPFIDX	Dispatcher	Invalid command in this state.	No action required.	Response Flags = 5'b11111 Next State = TIMER

Table 13: *Timer* State Transitions

2.7 CRTIMER Commands and SMC Almost Full

The SMC queues events to the Dispatcher. When this SMC to Dispatcher queue reaches a certain watermark, the SMC asserts the SMC_DISP_Almost_Full signal to the Dispatcher and the FDC. Asserting this signal indicates that no *new* events should be issued to the Processor Cores. This is part of a deadlock avoidance algorithm. For further information on why this signal is used the reader is directed to the *Deadlock Analysis and Avoidance* document.

Timer Events are considered *new* events since they are issued when a timer expires. To prevent Timer Events from being issued the FDC will respond with *No Space* if the SMC_DISP_Almost_Full signal is set and a CRTIMER command is received from the LUC with the FDC entry in the *CHECKED IN* state. By default the FDC will obey the SMC_DISP_Almost_Full signal. However, the DIS_SMC_DISP_BP bit of the CONTROL register can modify this behaviour.

Note that treating the CRTIMER command this way is better than having the Dispatcher not service the LUC Timer queue, since doing so would not allow the Dispatcher to service timer events that were issued before the backpressure was asserted.

2.8 FDC Statistics

The FDC keeps a small number of statistics regarding the number of commands received, number of times we ran out of CAM space etc. For more details the reader is directed to the register definitions of section 3.5, and specifically sections 3.5.6, 3.5.7 and 3.5.8.

2.9 Expected Performance

In this section we give an insight into the expected performance of this device. Note that this quick calculation is meant to guide implementation options, and is not meant to be a precise metric which can be used as a pass / fail test of the FDC. The true required performance will be guided by the various simulations.

We use multiple methods for estimating the FDC expected performance: the method that produces the highest performance requirement dictates the performance level of the FDC. Also note that these performance requirements are not all required at the same time. For example, the connections per second metric and the timer metric are disjoint: we use the connections per second metric or the time metric but not both at the same time.

2.9.1 FDC Performance Based on Connections Per Second Metric

This metric is based on a number of TCP connections being established and torn down with very little data transfer in between, e.g. lots of HTTP transactions.

Let us assume that processing a protocol event requires the following steps:

1. The Dispatcher receives the event, does a lookup flow key with create (LFKCREATE). Let us assume that no entry is found, and so one is created, and the free/busy data structures of section 2.4 are updated.
2. The Processor Core finishes processing this event, issues a "Done Event" to the Dispatcher, and the Dispatcher issues an update with FDC index (UPFIDX) command.
3. After the LUC has finished updating the flows state, it issues a remove with FDC index (RMFIDX) command.

The above commands are the minimal that are needed for processing a packet or interface event. For now we do not consider timers. Also note that on the last frame of a flow the tear down with FDC index (TDFIDX) command will be issued rather than a UPFIDX.

Of these three commands, only the LFKCREATE is an actual CAM lookup. The other commands (UPFIDX and RMFIDX) use direct indexes into the CAM.

Let us assume that the above three commands are issued for each event in a flow. Let us then state that when measuring connections per second, there will be eight such events per flow[21]. Therefore, if we require 500,000 connections per second, then we expect 4,000,000 events per second, i.e. we expect 4,000,000 [LFKCREATE, UPFIDX, RMFIDX] commands per second.

2.9.2 FDC Performance Based on Bulk Data Transfer Metric

This metric is based upon a small number of TCP connections that are established at start of day, and then used to transfer large amounts of data.

The current architecture uses two 10Gbit full duplex interfaces to the ACP. In one scenario we can envision one of these interfaces being used for the host CPU, the other for network traffic. Let us assume that

[21] This is assuming a push API with no automatic flow closing. For more details on the events in a flow see "TCP Processing Paths for the Content Processor" and "Queuing Model Trace of a Simple HTTP Request" from section 1.1.

maximum sized IEEE Ethernet frames are used for the bulk data transfer[22]. We also know that each Ethernet frame has an overhead of 20 bytes for inter packet gap etc. We will therefore receive (10G / ((1500 + 20) x 8) packets per second, which is approximately 822,000 packet events per second. Each event requires three FDC commands: LFKCREATE, UPFIDX and RMFIDX.

Suppose that for each packet event received, we also receive an interface read event to read the data contained in the TCP payload. Also suppose that this is a proxy connection, so the data is written back to another socket via an interface write event. We assume that this data is acknowledged on the input stream. So, for each packet event received we also assume an interface read event and an interface write event. For bulk transfer on a 10Gbit full duplex interface we will therefore receive approximately 2,466,000 events per second. For bulk data transfer the FDC must therefore execute 2,466,000 [LFKCREATE, UPFIDX, RMFIDX] commands per second.

2.9.3 FDC Performance Based on the Timer Metric

The Astute Content Processor has a goal of servicing the timer expiration of 500,000 flows in 200ms. Servicing a timer event is similar to servicing a packet or interface event, except that instead of using an LFKCREATE at the start of the event, we start with a [CRTIMER, SERVTIMER] combination. Since 500,000 timer expirations in 200ms is 2,500,000 timer expirations per second, the FDC must be able to execute 2,500,000 [CRTIMER, SERVTIMER, UPFIDX, RMFIDX] commands per second.

3 Interfaces

3.1 Dispatcher FDC Bus

The Dispatcher FDC Bus carries FDC commands for both the Dispatcher *and* the LUC.

3.1.1 Expected Performance

According to section 2.9.1, for each event in a connection the Dispatcher sends two FDC commands (LFKCREATE, UPFIDX) and the LUC sends one command (RMFIDX). Since there are eight events in a connection, and we are to achieve 500K connections per second, this is 12,000,000 Dispatcher to FDC commands (and responses) per second. Each command is 128-bits long, therefore requiring a bandwidth of 1.536Gbits per second. This can be achieved with an 8-bit bus at 266MHz, but to ease implementation we use a 128-bit wide bus.

3.1.2 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
Disp_FDC_CMD[127:0]	128	I	Dispatcher to FDC Data.
Disp_FDC_VAL	1	I	Dispatcher to FDC valid bit.
FDC_Disp_VAL_CLR	1	O	FDC acknowledges the Disp_FDC_CMD by clearing Disp_FDC_VAL

Table 14: Dispatcher FDC Bus Signals

3.2 FDC Dispatcher Bus

The FDC Dispatcher Bus carries FDC responses for both the Dispatcher *and* the LUC.

[22] Jumbo-sized Ethernet frames (16K) would put even less of a load on the Dispatcher.

3.2.1 Expected Performance

Using the same logic as section 3.1.1, there are 12,000,000 FDC to Dispatcher responses (and commands) per second. Each response is 39-bits long, therefore requiring a bandwidth of about 468Mbits per second. This can be achieved with a very narrow bus, but to ease implementation we use a 39-bit wide bus.

3.2.2 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
FDC_Disp_Resp[38:0]	39	O	FDC to Dispatcher Data.
FDC_Disp_Resp_EN	1	O	FDC to Dispatcher valid bit.

Table 15: FDC Dispatcher Bus Signals

3.3 MMC Bus

The reader is directed to the ManageMent and Control HLD for full details on the MMC bus.

3.4 Data Formats

The following sections define the commands and responses that are used on the various FDC buses.

3.4.1 Data Format Goals

1. The FDC request formats is to be as consistent as possible with the Event Format of the Dispatcher. For example, the Flow Key position should be such that it requires no bit movement from the Event to the FDC Command.

2. The FDC responses show the state of the FDC entry that was found (via the Flags field). Not all responses will require all these fields, but they are included for simplicity.

3. The FDC responses overload the Flags field such that the values 5'b11111 and 5'b11110 represent exceptional conditions rather than FDC states.

4. The response format should allow the majority of the FDC CAM entry to be copied directly into it[23].

3.4.2 Create Timer (CRTIMER) Formats

Figure 6 illustrates the format that should be used when issuing a CRTIMER command, and Figure 7 illustrates the associated response format.

If a timer entry is successfully created then the flags field of Figure 7 will indicate that the entries state is TIMER (10101). Any value of flags other than TIMER indicates that the command was not successful and must be re-issued.

[23] This is true except for the location of the Event Mode bits in the LFKCREATE and SERVTIMER responses.

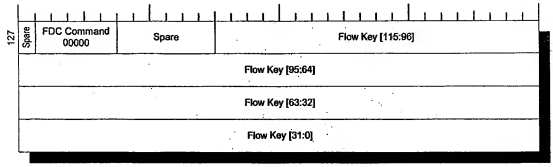

Figure 6: CRTIMER Request Data Format

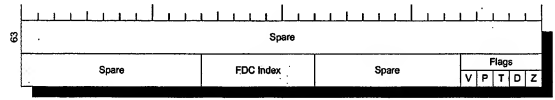

Figure 7: CRTIMER Response Data Format

3.4.3 Get Event (GETEVENT) Formats

Figure 8 illustrates the format that should be used when issuing a GETEVENT command, and Figure 9 illustrates the associated response format. The M bit indicates whether the Event Type of Event Mask should be used when executing the GETEVENT command. If the value of M is zero then the Event Type should be used, otherwise the Event Mask should be used. For more details on the use of the Event Type and Event Mask the reader is referred to section 2.4.6. Note that the Event Mask is a Core Bitmap format, as described in section 2.1.3.

If an event was successfully allocated then the *Alloc* bit of the response is set to one, and the *Event Index* and *Core ID* indicate the event element that was allocated. If the *Alloc* bit is set to zero then an event index could not be allocated.

Note that the response for a GETEVENT does not include an Event Mode field. This field is not required in the response since the Dispatcher will set the Event Mode in the Event to a fixed value.

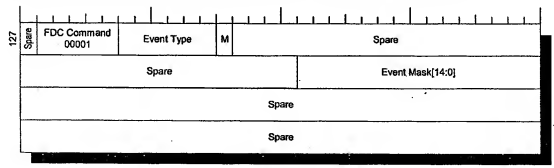

Figure 8: GETEVENT Request Data Format

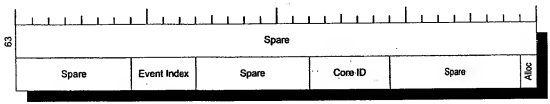

Figure 9: GETEVENT Response Data Format

3.4.4 Lookup with Flow Key and Create (LFKCREATE) Request Data Formats

Figure 10 illustrates the format that should be used when issuing a LFKCREATE command, and Figure 11 illustrates the associated response format. Note that the request must indicate both the Flow Key and the Event Type that should be used when allocating a Processor Core. For more details on how Event Types are used see section 2.4.

The Flags field of the response (Figure 11) can be one of the following values:

1. 11111, indicating that the FDC entry is in the DELETE state, and as such no event index, Processor Cores or workspace IDs were assigned.
2. 11110, indicating that there is either no space in the FDC, or no Processor Cores were available, or no workspace IDs were available, or no event index was available.
3. *TIMER* (10101), indicating that the FDC entry is in the TIMER state, and as such no event index, Processor Cores or workspace IDs were assigned.
4. *RECEIVED* (10000) or *PENDING* (11000), indicating that an event index, Processor Cores and workspace IDs were successfully assigned.

If the *New Entry* flag of Figure 11 is set then this indicates that the FDC entry was created in response to this LFKCREATE command, i.e. the FDC entry just entered the *RECEIVED* state from the *CHECKED IN* state. This is used by the Dispatcher to determine whether to send a LUC command or not.

The Event Mode field in the response of Figure 11 indicates if the event should be forwarded to the Protocol Core. See section 2.4.1 for more details. Note that this field will either indicate that the event should be forwarded to the protocol core or the interface core. It will **never** indicate that the event should be forwarded to neither.

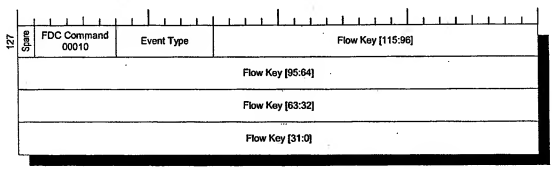

Figure 10: LFKCREATE Request Data Format

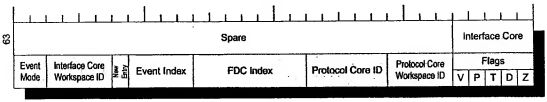

Figure 11: LFKCREATE Response Data Format

3.4.5 Release Event (RELEVENT) Data Formats

Figure 12 illustrates the format that should be used when issuing a RELEVENT command, and Figure 13 illustrates the associated response format. Although there is no response data required for this command, we still respond with an empty response format. This makes the FDC commands uniform as they then all supply responses. The purpose of the RELEVENT command is to mark the *Event Index* of the *Core ID* as available (see section 2.4.8).

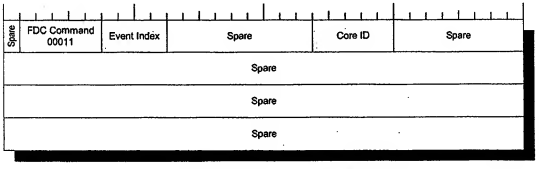

Figure 12: RELEVENT Request Data Format

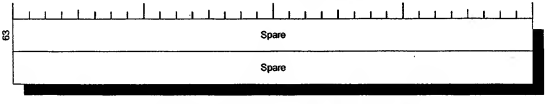

Figure 13: RELEVENT Response Data Format

3.4.6 Remove with FDC Index (RMFIDX) Data Formats

Figure 14 illustrates the format that should be used when issuing a RMFIDX command, and Figure 15 illustrates the associated response format. With reference to Figure 5, if the FDC entry is removed from the CAM then the Flags field will indicate the *CHECKED IN* state (00000). If the FDC entry could not be removed due to a new event being received for this flow, then the Flags field will indicate the *RECEIVED* state (10000). If the state of the FDC entry does not allow an RMFIDX command then the Flags field will indicate the value 5'b11111.

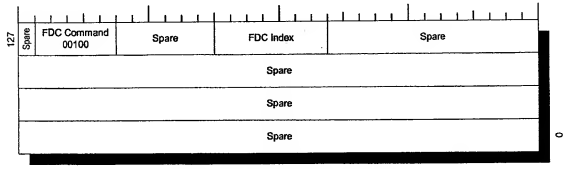

Figure 14: RMFIDX Request Data Format

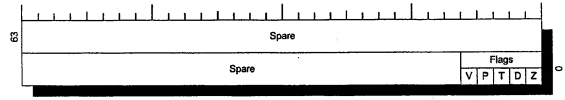

Figure 15: RMFIDX Response Data Format

3.4.7 Service Timer (SERVTIMER) Data Formats

Figure 16 illustrates the format that should be used when issuing a SERVTIMER command, and Figure 17 illustrates the associated response format. Note that the SERVTIMER command must include the Event Type that is to be used when allocating the Processor Cores. For more details on how Event Types are used see section 2.4.

If an Event Index, Processor Cores and Workspace IDs could be allocated then the Flags field of Figure 17 will indicate the RECEIVED state (10000). If the Flags field is any other value then an Event Index / Processor Cores / Workspace IDs could not be allocated, and the service timer request should be re-issued at a later time.

The Event Mode field in the response of Figure 17 indicates if the event should be forwarded to the Protocol Core. See section 2.4.1 for more details. Note that this field will either indicate that the event should be forwarded to the protocol core or the interface core. It will **never** indicate that the event should be forwarded to neither.

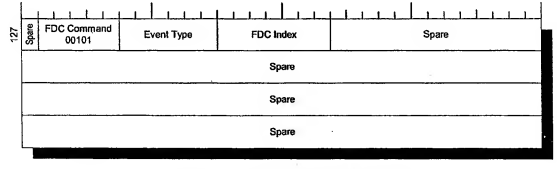

Figure 16: SERVTIMER Request Data Format

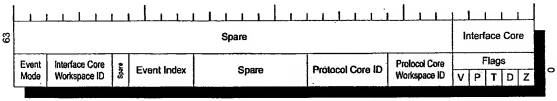

Figure 17: SERVTIMER Response Data Format

3.4.8 Tear Down with FDC Index (TDFIDX) Data Formats

Figure 18 illustrates the format that should be used when issuing a TDFIDX command, and Figure 19 illustrates the associated response format. The *Event Index* and *Event Mode* of the command, and the *Protocol Core ID* and *Interface Core ID* of the FDC entry indicate which event queue element is now available and the FDC should mark it as such (see section 2.4.8). If the event count of the FDC entry reaches zero then the workspace ID can also be released (see section 2.4.9).

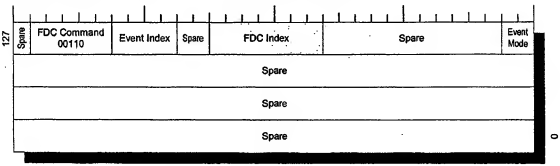

Figure 18: TDFIDX Request Data Format

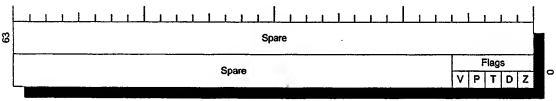

Figure 19: TDFIDX Response Data Format

3.4.9 Update with FDC Index (UPFIDX) Data Formats

Figure 20 illustrates the format that should be used when issuing a UPFIDX command, and Figure 21 illustrates the associated response format. The *Event Index* and *Event Mode* of the command, and the *Protocol Core ID* and *Interface Core ID* of the FDC entry indicate which event queue element is now available and the FDC should mark it as such (see section 2.4.8). If the event count of the FDC entry reaches zero then the workspace ID can also be released (see section 2.4.9).

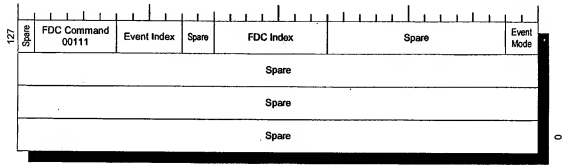

Figure 20: UPFIDX Request Data Format

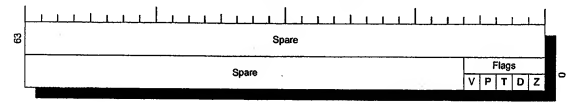

Figure 21: UPFIDX Response Data Format

3.5 Configuration Registers

3.5.1 Register Map

Table 16 defines the register map that the FDC uses. Note that the FDC does not have a direct MMC interface. Instead all FDC registers are accessed via the register block of the Dispatcher. The offsets in Table 16 are relative to the FDC block that is assigned in the Dispatcher HLD. The INV_MMC_ADDR bit of the Dispatcher STATUS register captures invalid register accesses for the FDC.

3.5.2 FDC Register Implementation

3.5.2.1 Write Access

During a FDC register write, no ACK is supplied to the Management Controller (MMC). The FDC must therefore be able to process back to back writes. According to the *Management Controller HLD*, the FDC must be able to process back to back writes at a rate of one every 7 clock cycles.

3.5.2.2 Read Access

During a FDC register read an ACK is supplied, so there is no issue regarding back to back reads.

Offset in Dispatcher Block (Hex)	Mode	Register Name	Description
0080	Read Only	STATUS	Status information, e.g. error notification bits.
0081	Read / Write	CONTROL	Control information, e.g. reset bit, dual/single core mode.
0082	Read / Write	MASK	Mask to apply before raising the Dispatcher interrupt.
0083	Read Only	CMD_CNT	Count of the number of FDC commands received.
0084	Read Only	NOCAM_CNT	Count of the number of times we ran out of CAM space.
0085	Read Only	NOPCORE_CNT	Count of the number of times a suitable Processor Core could not be found.
0086	N/A	SPARE	N/A
0087	Read / Write	CAM_ADDR	FDC CAM address register for CPU access.
0088-008C	Read Only	CAM_DATA	FDC CAM data registers for CPU access.
008D-008F	N/A	SPARE	N/A
0090	Read / Write	FREEBUSY0	Free/busy bits for the events and workspaces IDs of Core ID 0.
0091	Read / Write	FREEBUSY1	Free/busy bits for the events and workspaces IDs of Core ID 1.
0092	Read / Write	FREEBUSY2	Free/busy bits for the events and workspaces IDs of Core ID 2.
0093	Read / Write	FREEBUSY3	Free/busy bits for the events and workspaces IDs of Core ID 3.
0094	Read / Write	FREEBUSY4	Free/busy bits for the events and workspaces IDs of Core ID 4.
0095	Read / Write	FREEBUSY8	Free/busy bits for the events and workspaces IDs of Core ID 8.
0096	Read / Write	FREEBUSY9	Free/busy bits for the events and workspaces IDs of Core ID 9.
0097	Read / Write	FREEBUSY10	Free/busy bits for the events and workspaces IDs of Core ID 10.
0098	Read / Write	FREEBUSY11	Free/busy bits for the events and workspaces IDs of Core ID 11.
0099	Read / Write	FREEBUSY12	Free/busy bits for the events and workspaces IDs of Core ID 12.
009A	Read / Write	FREEBUSY16	Free/busy bits for the events and workspaces IDs of Core ID 16.
009B	Read / Write	FREEBUSY17	Free/busy bits for the events and workspaces IDs of Core ID 17.
009C	Read / Write	FREEBUSY18	Free/busy bits for the events and workspaces IDs of Core ID 18.
009D	Read / Write	FREEBUSY19	Free/busy bits for the events and workspaces IDs of Core ID 19.
009E	Read / Write	FREEBUSY20	Free/busy bits for the events and workspaces IDs of Core ID 20.
009F	N/A	SPARE	N/A
00A0-00BF	Read / Write	EVENT_MASK	Indicates which Processor Cores are available. Indexed by *Event Type* of the FDC command, allowing for the *Event Type* sub-structure of section 2.4.2.1.

Table 16: FDC Register Map

3.5.3 Status (STATUS) Register [0080H]

Indicates the status of the FDC. Note that all error bits are reset when read.

Default value: 0

Bits	Name	Description
0	UNREC_CMD_ERR	This bit is set if an unrecognised command is encountered in the *FDC Command* field of a FDC request. Note that this is the **only** condition when this bit is set. If such a command is received then the FDC responds with 5'b111111 in the *Flags* field.
1	EC_OVFLOW	This bit is set if the FDC attempts to increment an *Event Count* that has value 7'h7f, i.e. it is set if the *Event Count* of an FDC entry overflows. Should this bit be set, the FDC must be reset. We check for such a condition when processing an LFKCREATE in either the *RECEIVED* or *PENDING* states – those are the only times the *Event Count* can overflow. See section 2.2.1 for further details.
2-31	N/A	N/A

Table 17: Status Register Bit Definitions

3.5.4 Control (CONTROL) Register [0081H]

Default value: 0

Bits	Name	Description
0	CAM_INIT	This is set to 1 at reset. When set it will take 16 clock cycles to initialise the CAM, after which it will clear it self. Writing a 1 to this bit will re-initialise the CAM. This bit should only be set when all interfaces on the Dispatcher are disabled.
1	DUAL_CORE_MODE	Set to 1 if the FDC is to operate in dual core mode. Set to 0 for single core mode.
2	DIS_SMC_DISP_BP	This is the *Disable SMC-DISP Backpressure* bit. If this bit is set to zero then the SMC_DISP_Almost_Full signal is used to modify the response of CRTIMER commands in the *CHECKED IN* state. If this bit is set to one then CRTIMER commands in the *CHECKED IN* state are not modified. See section 2.7 for further details.
3-31	N/A	N/A

Table 18: Control Register Bit Definitions

3.5.5 Mask (MASK) Register [0082H]

Default value: 0

Bits	Name	Description
0-31	MASK	Mask to apply to the STATUS register before determining whether to raise the interrupt line to the Dispatcher. Not all bits in this register are used. Only the bits that have corresponding defined bits in the STATUS register are used.

Table 19: Mask Register Bit Definitions

3.5.6 Command Count (CMD_CNT) Register [0083H]

Default value: 0

Bits	Name	Description
0-31	CNT	Count of the number of FDC commands received. This counts commands from both the LUC and the Dispatcher. Cleared on read.

Table 20: Command Count Register Bit Definitions

3.5.7 No CAM Space Count (NOCAM_CNT) Register [0084H]

Default value: 0

Bits	Name	Description
0-31	CNT	How many times did we want to create a CAM entry but could not due to a lack of space. Cleared on read.

Table 21: No CAM Space Count Register Bit Definitions

3.5.8 No Processor Core Available (NOPCORE_CNT) Register [0085H]

Default value: 0

Bits	Name	Description
0-31	CNT	How many times could we not create an FDC entry since a suitable Processor Core could was not available. Cleared on read.

Table 22: No Processor Core Available Register Bit Definitions

3.5.9 CAM Address (CAM_ADDR) Register [0087H]

The CAM_ADDR and CAM_DATA registers are used for debug (read-only) access to the contents of the FDC. For read access the microprocessor would make the following operations:

1. Write the address to be read into the CAM_ADDR register.
2. Read the CAM_DATA_0 register. It is on this operation that the FDC triggers a read of the CAM. It must latch the result of this read into the CAM_DATA registers.
3. Read the CAM_DATA_1 through CAM_DATA_4 registers to obtain the rest of the CAM entry.

Write access to the FDC CAM is not provided.

Default value: 0

Bits	Name	Description
0-6	ADDR	The address of the CAM entry that is to be read. Addresses range from 0 up to the maximum number of entries in the FDC minus one.
7-31	N/A	N/A

Table 23: CAM Address Register Bit Definitions

3.5.10 CAM Data (CAM_DATA) Registers [0088H – 008CH]

There are five CAM_DATA registers in total, named CAM_DATA_0 through CAM_DATA_4. They are arranged such that the CAM_DATA_0 register at the lowest FDC address contains bits 0-31 of the data with reference to the FDC format of Figure 3. The next CAM_DATA register, CAM_DATA_1 contains bits 32-63 of data with reference the FDC format. This continues until the last CAM_DATA register, CAM_DATA_4, which contains bits 128-159. For more information on how to use these registers see section 3.5.9 about the CAM_ADDR register.

Default value: 0

Bits	Name	Description
0-31	DATA	Data from a Flow Directory CAM read.

Table 24: CAM Data Register Bit Definitions

3.5.11 Free/Busy (FREEBUSY0 through FREEBUSY20) Registers [0090H – 009EH]

The FREEBUSY registers are an array of 32-bit registers, with one register per Processor Core. This register block is non-contiguous in the sense that the Core ID is **not** used to index into it, i.e. Core ID 9 uses FREEBUSY9 but it is not the 9th FREEBUSY register from the base of the FREEBUSY register block, it is the 7th.

See sections 2.4.10 and 2.4.11 for information on initializing EBITS and WBITS. Reading from these registers is for debug only, and indicates which event queue elements and workspace IDs are currently free/busy.

The FDC actually uses two registers to track the Free/Busy information. When a processor writes to a FREEBUSY register the FDC duplicates that value and stores it in another INIT_EBITS register. This INIT_EBITS value is the `init_event_bits` value that was used in section 2.4.4. There is one INIT_EBITS register for each FREEBUSY register. The INIT_EBITS register is used by the FDC to determine the event queue length, allowing it to wrap the `head` value of section 2.4.4 correctly. For example, if the EBITS value that is written is 8'b0011111 then that indicates that there are a total of five elements in the event queue. During a read of the FREEBUSY register, the INIT_EBITS value is accessible via the INIT_EBITS field. Note that this is only during a FREEBUSY register read. Those bits are ignored on a write.

Note that the EBITS must be written such that all the 1's are flush to the right, i.e. it cannot contain holes. The EBITS format uses Event Numbers as opposed to Event Indexes (see section 2.1.4).

Default value: See field descriptions

Bits	Name	Description
0-7	EBITS	These are the `event_bits` of section 2.4. See section 2.4.11 on how to initialise these bits dependent upon the event queue depth. Note that these are in the Event Number format, and are not Event Indexes. See section 2.1.4 for clarification. Default value: 8'hf.
8-15	INIT_EBITS	These bits are ignored during a write of this register. On a read these bits return the value of the INIT_EBITS register, as described above. Default value: 8'hf.
16-31	WBITS	These are the `workspace_id` bits of section 2.4. See section 2.4.10 on how to initialise these bits dependent upon the workspace size. Default value: 16'hf.

Table 25: Free/Busy Register Bit Definitions

3.5.12 Event Mask (EVENT_MASK) Registers [00A0H – 00BFH]

There are thirty-two Event Mask registers. The Event Mask registers are indexed using the lower 5-bits of the Event Type, as described in section 2.4.2.1. A write to one of these registers sets the `pc_event_mask` and `ic_event_mask` of section 2.4. Reading from one of these registers will return the current value of the `pc_event_mask` and `ic_event_mask` values for that Event Type.

Note that for the same Event Type value, IC_EVENT_MAKS AND PC_EVENT_MASK must equal zero. This ensures that in dual core mode the same Processor Core cannot be selected as both a Protocol Core and an Interface Core. If the same Processor Core is required to perform both Protocol Core and Interface Core functionality then the FDC should be set to single core mode.

All the EVENT_MASK registers should be initialised, even if that event type will not be used. If the event type is not expected to be used then simply select a default Processor Core to send the event to, and program the Dispatcher so that the forwarding mode for this event type is *Unicast Event Processing, Drop Condition*. Failure to do this will result in an errant event type becoming stuck at the front of a Dispatcher queue since it will appear that no Processor Core is available to service it.

Default value: See field descriptions

Bits	Name	Description
0-14	PC_EVENT_MASK	If bit P is set then Protocol Core P is available in this core group. This mask is in the Core Bitmap format, as described in section 2.1.3. Default value: 15'h7fff.
15-29	IC_EVENT_MASK	If bit I is set then Interface Core I is available in this core group. This mask is in the Core Bitmap format, as described in section 2.1.3. Default value: 0.
30	ALLOC_PC_EVENT	If set to 1 then this event type requires that an event queue element be allocated from a Processor Core in the PC_EVENT_MASK bitmap. If set to 0 then an event queue element should be allocated from a Processor Core in the IC_EVENT_MASK bitmap. Note that if the FDC is programmed to operate in Single Core Mode (see DUAL_CORE_MODE of CONTROL register) then the software **must** set the ALLOC_PC_EVENT bit to value 1 for **all** Event Types. This forces the PC_EVENT_MASK to be used in Single Core Mode. Default value: 0.
31	N/A	N/A

Table 26: Event Mask Register Bit Definitions

3.6 Initialisation

The following sequence should be used to initialise the FDC from the reset state:

1. The CAM is automatically cleared when the FDC is brought out of reset, so no extra action is required to clear the CAM.

2. Program the following FDC registers with values appropriate to the software:
   a. Control (CONTROL) register (section 3.5.4). The DUAL_CORE_MODE of the CONTROL register should be set according to the desired mode of operation.
   b. Mask (MASK) register (section 3.5.5).
   c. Free/Busy (FREEBUSY0 through FREEBUSY20) register block (section 3.5.11).
   d. Event Mask (EVENT_MASK) register block (section 3.5.12). As described in section 3.5.12, it is important to initialise all of these registers.

3. Done. The FDC is now initialised.

4 Open Issues

None.

5 Summary

The preceding sections should have accurately described the operation of the Flow Director CAM. Please notify the author of any discrepancies, omissions or typos.